CENTRO INTERNACIONAL DE ESTUDOS
DE DOUTORAMENTO E AVANZADOS
DA USC (CIEDUS)

TESE DE DOUTORAMENTO

ACQUISITION AND DECLARATIVE ANALYTICAL PROCESSING OF SPATIO-TEMPORAL OBSERVATION DATA

Sebastián Villarroya Fernández

ESCOLA DE DOUTORAMENTO INTERNACIONAL

PROGRAMA DE DOUTORAMENTO EN INVESTIGACIÓN EN TECNOLOXÍAS DA INFORMACIÓN

SANTIAGO DE COMPOSTELA

ANO 2018

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

DECLARACIÓN DO AUTOR DA TESE

Acquisition and Declarative Analytical Processing

of Spatio-Temporal Observation Data

D. Sebastián Villarroya Fernández

Presento miña tese, seguindo o procedemento adecuado ao Regulamento, e declaro que:

1) A tese abarca os resultados da elaboración do meu traballo.
2) No seu caso, na tese se fai referencia as colaboracións que tivo este traballo.
3) A tese é a versión definitiva presentada para a súa defensa e coincide ca versión enviada en formato electrónico.
4) Confirmo que a tese non incorre en ningún tipo de plaxio de outros autores nin de traballos presentados por min para a obtención de outros títulos.

En Santiago de Compostela, 20 de Xullo de 2018

Asdo. Sebastián Villarroya Fernández

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

AUTORIZACIÓN DO DIRECTOR / TITOR DA TESE

Acquisition and Declarative Analytical Processing

of Spatio-Temporal Observation Data

D. José Ramón Ríos Viqueira
D. José Manuel Cotos Yáñez

INFORMA/N:

Que a presente tese, correspóndese co traballo realizado por D. ***Sebastián Villarroya Fernández****, baixo a miña dirección, e autorizo a súa presentación, considerando que reúne os requisitos esixidos no Regulamento de Estudos de Doutoramento da USC, e que como director desta non incorre nas causas de abstención establecidas na Lei 40/2015.*

En Santiago de Compostela, 20 de Xullo de 2018

Asdo. José Ramón Ríos Viqueira

Asdo. José Manuel Cotos Yáñez

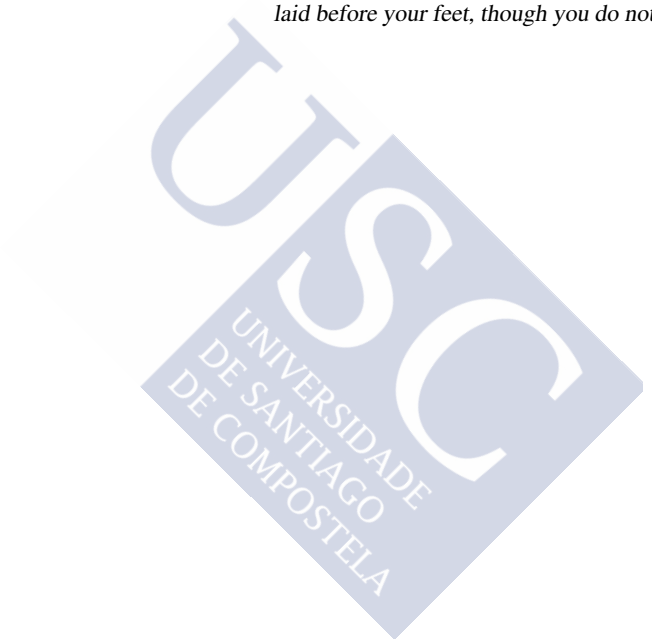
USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

A Sabela, Álex y Sandra

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Maybe the paths that you each shall tread are already laid before your feet, though you do not see them.

Lady Galadriel



USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Agradecimientos

Ha llegado el final de una etapa importante en mi formación académica, la más importante hasta el momento sin duda. Y ha sido un camino duro y largo, muy largo, quizás demasiado. Con muchos momentos de incertidumbre, desesperanza y temor a estar caminado sin rumbo ni destino. Sin embargo, ha sido una experiencia enriquecedora a nivel personal. Puedo asegurar, sin miedo a equivocarme, que me ha hecho más fuerte, tenaz, perseverante, seguro y, quiero pensar, mejor persona. Además, durante este período han ocurrido los dos momentos más felices de mi vida. Han sido tan felices que hasta les hemos puesto nombre: Álex y Sabela. Por tanto, no tengo más que palabras de agradecimiento por haber podido vivir esta experiencia.

Me gustaría dar las gracias a muchas personas y organizaciones que han hecho que este camino sea mucho más fácil. A todos ellos les estaré eternamente agradecido. Sin embargo, todos entenderán que quiera tener un recuerdo especial para algunos de ellos.

En primer lugar, quiero dar las gracias a mis directores de tesis. A José Ramón, por confiar en mí y en que sería capaz de llevar este proyecto a buen puerto. Por todo el apoyo en los momentos difíciles de la tesis, que los ha habido, y por hacer camino dónde no lo había. Pero sobre todo, por ser mucho más que un director de tesis, por llevar su ayuda y amistad mucho más allá. A Manel, por haber confiado en mí también, por haberme traído aquí. Sin su ayuda incondicional yo no habría llegado hasta el final de este camino.

Quiero agradecer a mi familia todo el sacrifio que han tenido que hacer para que tú puedas estar leyendo esto ahora. No serían suficientes todas las páginas de esta tesis si tuviese que escribir todas y cada una de las cosas que le tengo que agraceder a Sandra. Su apoyo, dedicación, comprensión, paciencia, ánimo, confianza... créeme, la lista no tiene fin. Y todo ello sin ser justamente correspondido. A mis dos grandes amores, Álex y Sabela, por todas las

horas robadas. Dedicadas a este trabajo en lugar de haberlas disfrutado a su lado. Es una deuda no se puede saldar, impagable.

También me quiero acordar de mis padres. Por todo el sacrificio que tuvieron que hacer para darme una formación sin la cual yo no estaría escribiendo esto. Por la educación que me dieron. Por tantas cosas. A mi madre, por ser una luchadora, por no rendirse nunca, por vencer lo invencible. A mi padre, que aunque nunca fue una persona de dar lecciones me enseñó la más importante de todas: amar la vida por encima de todas las cosas.

A mis abuelos, aunque no estén relacionados directamente con esta tesis siempre han tenido palabras de apoyo para absolutamente todo lo que he querido hacer.

A los compañeros del grupo de investigación COGRADE. Por hacerme sentir uno más del grupo desde el primer momento. Por todos los buenos momentos en el trabajo y, sobre todo, fuera de él. Y por toda la ayuda prestada tantas y tantas veces. Esto no sería lo mismo sin vosotros.

Al Centro Singular de Investigación en Tecnologías de la Información (CiTIUS). Por toda la ayuda administrativa prestada. Por la ayuda técnica en muchos proyectos. Y, sobre todo, por cofinanciar mi asistencia al *1st Summer School on Data Science*, organizado por ACM Sigmod.

A la Diputación de A Coruña. Por concederme la Beca de Investigación 2012.

Al Centro de Supercomputación de Galicia (CESGA). Por toda la infraestructura y ayuda prestadas en la ejecución de los experimentos de esta tesis. Y especialmente quiero dar las gracias a Javier Cacheiro López.

Finalmente, quiero dar las gracias a las organizaciones e instituciones que apoyaron, contribuyeron o financiaron los siguientes proyectos de investigación relacionados con esta tesis:

– *Patrimonio cultural de la Eurorregión Galicia-Norte de Portugal: Valoración e Innovación. GEOARPAD (0358_GEOARPAD_1_E)*. Programa INTERREG V-A España-Portugal (POCTEP) 2014-2020. Fondo Europeo de Desarrollo Regional (FEDER). Unión Europea.

– *FUTURE-HDA: Internet del Futuro en el Hogar Digital Asistencial (ITC-20113075)*. CDTI, Programa FEDER-INNTERCONECTA.

– *Desarrollo de un servicio de análisis espacial y su aplicación en la implementación de un sistema de gestión de hábitats humanos (TIN2010-21246-C02-02)*. Plan Nacional del Ministerio de Ciencia e Innovación.

– *Proyecto Minieólica: Fomento de la tecnología eólica de pequeña potencia. Subproyecto 3.4: Evaluación y diseño de un proyecto demostrador de energía eólica e hidrógeno (PS-120000-2006-5).* Proyecto Singular Estratégico del Ministerio de Ciencia e Innovación.

– *Proyecto Peixe Verde. Subproyecto 1: Toma de Datos (PSE-370300-2007-1).* Proyecto Singular Estratégico del Ministerio de Educación y Ciencia.

– *Rede de Tecnoloxías Cloud e Big Data para HPC (R2014-049).* Xunta de Galicia.

– *Sistema de Información Xeográfica para a xestión e difusión da información meteorolóxica e oceanográfica de Galicia. Subproxecto USC (09MDS034522PR).* Plan Galego, Xunta de Galicia.

July 2018

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Acknowledgments

The end of an important stage in my academic education has come, the most important one. It has been a hard and long journey, very long, maybe too much. With many moments of uncertainty, despair and fear of being walking without destination. However, it has been an enriching personal experience. I become a stronger, more persevering, self-confident and, hopefully, better person. In addition, the two happiest moments of my life happened during this period. The birth of my children: Alex and Sabela. Therefore, I am extremely grateful for having lived this experience.

I would like to thank many people and organizations that have made this road much easier. I will be eternally grateful to all of them. However, everyone will understand that I want to have a special memory for some of them.

First of all, I want to thank my thesis supervisors. Prof. José Ramón Ríos Viqueira, for trusting me and believing that I would be able to lead this project to a successful destination. For all his support in the hard moments during this thesis work and for making the way where there was none. But above all, for being much more than a thesis supervisor, for bringing his help and friendship much further. To Prof. José Manuel Cotos Yáñez, for trusting me too, for bringing me here. Without his unconditional help I would not have reached the end of this path.

I want to thank my family for the huge sacrifice they have made. I would need more pages to write all of the things that I want to thank Sandra than to write this thesis dissertation. Her support, dedication, understanding, patience, encouragement, trust ... believe me, the list has no end. And all this without being justly reciprocated. To my two great loves, Alex and Sabela, for all the stolen hours. Dedicated to this work instead of having enjoyed with them. It is a debt that can not be settled.

I also want to remember my parents. For all the sacrifice they had to do to give me a good academic training. For the education they gave me. For so many things. To my mother, for being a fighter, for never giving up, for overcoming the unbeatable. To my father, who taught me the most important lesson: to love life above all things.

To my grandparents. Although they are not directly related to this thesis, they have always supported everything I have wanted to do.

To the colleagues of the COGRADE research group. For making me feel an important part of the group from the beginning. For all the good times at work and, specially, outside of it. And for all the help given to me so many times. This would not be the same without you.

To the Research Center on Information Technologies (CiTIUS). For all the administrative help provided. For the technical help in many projects. And, above all, for co-funding my attendance at the *1st Summer School on Data Science*, organized by ACM Sigmod.

To the Diputación de A Coruña. For granting me the 2012 Research Scholarship.

To the Galicia Supercomputing Center (CESGA). For all the infrastructure and help provided during the execution of the experiments of this thesis. And especially I want to thank Javier Cacheiro López.

Finally, I want to thank the organizations and institutions that supported, contributed or funded the following research projects related to this thesis:

– *Patrimonio cultural de la Eurorregión Galicia-Norte de Portugal: Valoración e Innovación. GEOARPAD (0358_GEOARPAD_1_E)*. INTERREG V-A España-Portugal (POCTEP) Program, 2014-2020. European Regional Development Fund (ERDF). European Union.

– *FUTURE-HDA: Internet del Futuro en el Hogar Digital Asistencial (ITC-20113075)*. Center for the Development of Industrial Technology (CDTI) and FEDER-INNTERCONECTA Program.

– *Desarrollo de un servicio de análisis espacial y su aplicación en la implementación de un sistema de gestión de hábitats humanos (TIN2010-21246-C02-02)*. National Research Program, Ministry of Science and Innovation.

– *Proyecto Minieólica: Fomento de la tecnología eólica de pequeña potencia. Subproyecto 3.4: Evaluación y diseño de un proyecto demostrador de energía eólica e hidrógeno (PS-120000-2006-5)*. Ministry of Science and Innovation.

– *Proyecto Peixe Verde. Subproyecto 1: Toma de Datos (PSE-370300-2007-1)*. Ministry of Education and Science.

– *Rede de Tecnoloxías Cloud e Big Data para HPC (R2014-049)*. Xunta de Galicia.

– *Sistema de Información Xeográfica para a xestión e difusión da información meteorolóxica e oceanográfica de Galicia. Subproxecto USC (09MDS034522PR)*. Xunta de Galicia.

July 2018

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Contents

Abstract

A myriad of data acquisition devices is observing every day more variables and generating a vast amount of data in almost every application domain. Environmental observation data is an essential portion of such generated data, whose spatio-temporal nature has posed interesting challenges in the area of Environmental Observation Data Management Systems. Two features are common to all these systems: spatio-temporal observations and heterogeneity. In the context of this Thesis, the Observations and Measurements (O&M) conceptual schema was adopted as the theoretical framework for the definition of the concept of observation. Heterogeneity specifically concerns the data acquisition part of the aforementioned systems, which need to access data produced by heterogeneous sensing following different software/hardware specifications that are accessed through several communication protocols. A major challenge is to provide the required flexibility to enable data acquisition from heterogeneous sensing devices and data dissemination through heterogeneous end-user applications. The system must provide simple and straightforward mechanisms for the incorporation of the following components: 1) new in-situ sensing devices, 2) new data dissemination services, and 3) different persistent data storage technologies. Focusing on observation data management, a system must provide the following general functionalities to effectively manage observation data: 1) management of conventional Entity/Relationship data related to non-observed properties of entities, 2) management of sampled data over temporal, spatial (1D and 2D) and spatio-temporal domains, 3) Support for observation data semantics, and 4) efficient implementation for large scale shared-nothing distributed hardware architectures.

Moreover, the INSPIRE Directive of the European Union encourages the creation of a Spatial Data Infrastructure (SDI) to ensure the interoperability of spatial information systems in Europe. The application of INSPIRE in the Spanish legislative system forces public administrations to make their geographic data available through SDI services. Therefore, the new

enriched geographical knowledge allows for the appearance of many applications in different areas of knowledge that require spatial analysis capabilities.

In spite of the above needs, to the best of my knowledge, none of the available technologies and approaches found in data acquisition and data management literature provide support for all the aforementioned functionalities.

Therefore, the main objective of this Thesis is the design and implementation of a generic framework for spatio-temporal observation data acquisition and declarative analytical processing. This overall goal can be divided into three independent specific objectives:

- Design and implementation of a generic observation data acquisition and dissemination server.
- Design of a framework for declarative spatio-temporal analysis in very large spatio-temporal data warehouses.
- Efficient implementation of spatio-temporal on-line analytical processing in large scale distributed shared-nothing hardware architectures.

The main contributions of this Thesis may be summarized as follows:

- Generalization of a data acquisition and dissemination server, with great applicability in many scientific and industrial domains, providing flexibility in the incorporation of different technologies for data acquisition, data persistence and data dissemination.
- Definition of a new hybrid logical-functional paradigm to formalize a novel data model for the integrated management of entity and sampled data.
- Definition of a novel spatio-temporal declarative data analysis language for the previous data model.
- Definition of a data warehouse data model supporting observation data semantics, including application of the above language to the declarative definition of observation processes executed during observation data load.
- Column-oriented parallel and distributed implementation of the spatial analysis declarative language. The huge amount of data to be processed forces the exploitation of current multi-core hardware architectures and multi-node cluster infrastructures.

Resumen

Una enorme cantidad de dispositivos de adquisición de datos observan cada día más variables y generan ingentes cantidades de datos en la práctica totalidad de dominios de aplicación. Al mismo tiempo, cada día más áreas de investigación centran sus esfuerzos en la adquisición y gestión eficiente de los datos (p. ej., Redes de Sensores, Internet de las Cosas), y en el aprovechamiento inteligente de la información (p. ej., Minería y Análisis de datos).

Los datos de observaciones medioambientales constituyen una parte fundamental de dichos datos y, debido a su naturaleza espacio-temporal, presentan algunos desafíos interesantes en el área de la gestión de datos. De hecho, durante las últimas décadas se ha realizado un gran esfuerzo en la investigación de Sistemas de Gestión de Datos de Observaciones Medioambientales. Dichos sistemas presentan dos características comunes: observaciones espacio-temporales y heterogeneidad.

Observaciones espacio-temporales

La localización de cada observación en un determinado espacio de referencia y el instante temporal en el que el valor observado de una observación se aplica a la propiedad observada son elementos fundamentales de los metadatos, imprescindibles durante la ejecución del análisis. En el contexto de esta Tesis, el esquema conceptual definido por el estándar Observations & Measurements (O&M) del Open Geospatial Consortium (OGC) ha sido adoptado como marco teórico para la definición del concepto de *observación* y otros conceptos relacionados (p. ej., *propiedad observada*, *valor observado*). Una *observación* contiene un *valor observado* y los metadatos que proporcionan la semántica de observación necesaria para interpretarlo correctamente. Así, por ejemplo, un *valor observado* (25) con una *unidad de medida* específica (ºC) de una *propiedad observada* (temperatura) está proporcionado por una determinada *entidad observada* (estación_meteorológica). Una *entidad observada* puede

tener tanto propiedades convencionales (nombre, propietario) como *propiedades observadas* (temperatura, humedad). Los valores de las propiedades convencionales son asignados generalmente por alguna autoridad mientras que los valores de las *propiedades observadas* son estimados por un *proceso de observación* (sensor_temperatura). Los *procesos de observación* pueden ser de naturaleza muy diversa, p. ej., sensores físicos, tareas realizadas por operarios, algoritmos de procesamiento de datos. Por lo tanto, es obligatorio el registro de propiedades del *proceso de observación* específico utilizado para generar el *valor observado*. Es obligatorio también el registro del *phenomenonTime*, es decir, el instante temporal en el que el *valor observado* se aplica a la *propiedad observada*. El tipo de dato de observación producida por el *proceso de observación* viene determinado por dos características del propio proceso:

- Si el proceso se ejecuta de forma periódica, es decir, si es disparado por tiempo (time-triggered), o si el proceso es disparado por un evento específico (event-triggered).
- La posición relativa del proceso respecto a la entidad observada (in-situ o remoto).

Tipo de disparo

Los procesos disparados por evento (event-triggered) comienzan en un instante temporal determinado por un evento específico. Por ejemplo, una imagen LIDAR (Light Detection and Ranging) capturada en un instante determinado. Los procesos disparados por tiempo (time-triggered) se ejecutan con una frecuencia temporal predeterminada produciendo muestreos regulares en el dominio temporal. Un ejemplo sería un registro de valores de temperatura obtenidos por un sensor de una estación meteorológica cada diez minutos.

Localización relativa del sensor

Los sensores in-situ están ubicados en la misma posición donde se producen los valores de la *propiedad observada*. Se genera una única *observación* en cada instante temporal. Podemos mencionar como ejemplo un dispositivo GPS instalado en un vehículo. Los sensores remotos están ubicados lejos de donde se producen los valores de la *propiedad observada*. Se generan varios *valores observados* en cada instante temporal. Un ejemplo de sensor remoto es el Sonic Detection and Ranging (SODAR) utilizado para registrar la velocidad del viento a diferentes altitudes mediante la medición de la dispersión de ondas sonoras producida por turbulencia atmosférica. El SODAR genera un muestreo 1D de la velocidad del viento a lo

largo de posiciones discretas consecutivas sobre un perfil lineal vertical. Un ejemplo de sensor remoto instalado en una plataforma móvil es el Visible Infrared Imaging Radiometer Suite (VIIRS) del satélite Suomi NPP. Este sensor proporciona imágenes de la superficie terrestre con una resolución espacial de 750 metros. Entre los datos generados por este sensor podemos encontrar muestreos 2D regulares (llamados Rasters) del color y temperatura de la superficie del océano.

Heterogeneidad

La parte de los sistemas de gestión de datos encargada específicamente de la adquisición de datos presenta una gran heterogeneidad debido a dos factores fundamentales:

- Adquisición de datos heterogéneos producidos por dispositivos de adquisición de diferente naturaleza (p. ej., radar, lidar, GPS).
- Implementación de diferentes protocolos de comunicación (p. ej., RS-485, WiFi, Ethernet).

Las arquitecturas de los sistemas desarrollados tanto para aplicaciones de monitorización y adquisición de datos como para sistemas de control supervisado sin requisitos de tiempo real están compuestas mayoritariamente por tres elementos:

- *Aplicaciones de Usuario*: se encargan de realizar el análisis y visualización de los datos.
- *Dispositivos sensores*: ejecutan los procesos de observación y son muy heterogéneos tanto en su funcionalidad como en los protocolos de comunicación que implementan.
- *Servidores de Datos*: actúan como pasarelas entre dominios heterogéneos de *Aplicaciones de Usuarios* y colecciones heterogéneas de *Dispositivos Sensores*, homogeneizando el acceso a los datos.

Durante el diseño e implementación de los Sistemas de Gestión y Adquisición de Datos de Observaciones Medioambientales surgen diferentes problemas que suponen un desafío para la adquisición y gestión de datos de observaciones.

Respecto a la adquisición de datos de observaciones, cualquier esfuerzo encaminado a la generalización de los *Dispositivos Sensores* y las *Aplicaciones de Usuario* resulta ser infructuoso. En el caso de los *Dispositivos Sensores* es debido al fuerte condicionamiento respecto

a las especificaciones de los vendedores. Por su parte, las *Aplicaciones de Usuario* tienen una gran dependencia tanto del dominio específico de aplicación como de las preferencias de los usuarios. Por otro lado, tanto la funcionalidad como la arquitectura de los *Servidores de Datos* suelen ser muy similares en la inmensa mayoría de aplicaciones. Sin embargo, proporcionar la flexibilidad necesaria en la adquisición de datos desde *Dispositivos Sensores* heterogéneos y en la publicación de datos hacia *Aplicaciones de Usuario* heterogéneas es un gran desafío. Así pues, el sistema debe proporcionar mecanismos simples y directos que permitan la incorporación de los siguientes componentes:

- Nuevos *Dispositivos Sensores* in-situ.
- Nuevos Servicios de Diseminación de Datos.
- Diferentes tecnologías para el almacenamiento persistente de datos, adaptadas a los diferentes tipos de datos observados.

En cuanto a la gestión de datos de observaciones, de las diferentes soluciones existentes en el estado del arte se pueden extraer las siguientes funcionalidades que permiten la gestión eficiente de datos de observaciones:

- Gestión de datos Entidad/Relación (E/R) convencionales relacionados con propiedades de Entidades.
- Gestión de datos muestreados en los dominios temporal, espacial (1D y 2D) y espacio-temporal.
- Soporte para semántica de datos de observaciones. Se deben proporcionar los metadatos necesarios para poder representar las propiedades observadas de las Entidades.
- Implementación eficiente sobre arquitecturas hardware distribuidas a gran escala.

Los desarrollos y tendencias recientes en el campo de las arquitecturas software de datos espaciales para Sistemas de Información Geográfica (SIG) proponen la descomposición de los sistemas en servicios simples y bien definidos, habitualmente basados en la Web y cuyas interfaces sigan los estándares internacionales de interoperabilidad del OGC y de la International Organization for Standardization (ISO). De esta forma, Infraestructuras de Datos Espaciales (IDE) integradas por servicios distribuidos en Internet pueden ser puestas a disposición de desarrolladores SIG.

Además, se están adoptando importantes políticas de mejora de la disponibilidad de conjuntos de datos espaciales generados por diferentes administraciones públicas. En particular, la directiva INSPIRE de la Unión Europea incentiva la creación de una IDE que asegure la interoperabilidad de los sistemas de información espacial en Europa. La aplicación de dicha directiva en el sistema legislativo español fuerza a las administraciones públicas a publicar sus datos geográficos a través de servicios IDE. Este nuevo entorno de conocimiento geográfico favorece la aparición de muchas aplicaciones en diferentes áreas de conocimiento que requieren de habilidades en análisis espacial.

Sin embargo, a pesar de todo este entorno favorable, ninguna de las tecnologías y soluciones que se pueden encontrar en la literatura de adquisición y gestión de datos soportan todas las funcionalidades mencionadas anteriormente.

La mayor parte de sistemas de adquisición proporcionan gran flexibilidad a la hora de obtener datos de observaciones desde dispositivos sensores heterogéneos, pero carecen de la flexibilidad de almacenamiento y diseminación de datos requerida. Algunos sistemas propuestos proporcionan mecanismos flexibles en la incorporación de nuevos dispositivos sensores. Sin embargo, presentan grandes limitaciones o directamente carecen de flexibilidad a la hora de extender las tecnologías de almacenamiento de datos que implementan. La mayor parte de sistemas carecen o presentan una flexibilidad muy limitada a la hora de añadir nuevos servicios de diseminación.

En la literatura de sistemas de gestión de datos se puede encontrar una cantidad enorme de trabajos orientados a la gestión de datos de observaciones. De hecho, el área de bases de datos espaciales es una de las áreas de investigación más activas proporcionando una enorme cantidad de soluciones. Incluso el estándar SQL de ISO, implementado por la inmensa mayoría de sistemas gestores de bases de datos, ha sido extendido para proporcionar funcionalidades espaciales. Actualmente, estas herramientas permiten la consulta declarativa de datos espaciales, incluidos los Raster 2D. Las herramientas NoSQL y Data Warehouse implementan extensiones espaciales, aunque no soportan datos Raster. Las soluciones SIG disponibles actualmente proporcionan la funcionalidad necesaria para almacenar y procesar datos convencionales y espaciales, incluidos los datos Raster. Sin embargo, no dan soporte para el análisis declarativo de datos, que sí está soportado en los gestores de arrays de datos diseñados para el procesamiento de grandes colecciones de arrays de datos Raster. Aunque la especificación de análisis de datos relacionales de forma declarativa utilizando estructuras de arrays de datos es bastante difícil, se han presentado soluciones que intentan integrar la gestión de arrays y datos

relacionales. En este caso, el usuario está obligado a trabajar con dos semánticas diferentes, una para datos relacionales y otra para arrays. Existen también algunos sistemas diseñados para el análisis declarativo de streams de datos de sensores, pero no dan soporte para datos Raster. Finalmente, la semántica de datos de observaciones solo está proporcionada por los estándares del OGC, por la iniciativa Sensor Web Enablement (SWE) y por ontologías y modelos de datos de observaciones específicos. Pero ninguna de estas soluciones permite el análisis declarativo de datos de observaciones.

Basándonos en las características expuestas de los sistemas y soluciones disponibles actualmente, el objetivo principal de esta Tesis es el diseño e implementación de una herramienta genérica para la adquisición y procesado analítico declarativo de datos de observaciones espacio-temporales. Este objetivo global se puede dividir en tres objetivos específicos, que se detallan a continuación.

***Objetivo 1* (GeoDADIS):** Diseño e implementación de un servidor genérico para la adquisición y diseminación de datos de observaciones. Esta herramienta debe proporcionar la funcionalidad necesaria para presentar las siguientes características:

- Adquisición in-situ de datos de observación mediante canales de datos síncronos y asíncronos.
- Diseminación de datos de observaciones mediante servicios de datos cliente/servidor y publicación/suscripción.
- Incorporación simple y directa de nuevas tecnologías de almacenamiento de datos de observaciones.

***Objetivo 2* (SODA):** Diseño de una herramienta para el análisis espacio-temporal declarativo en almacenes muy grandes de datos espacio-temporales. El concepto matemático de función es la base de un nuevo modelo de datos que integra datos Raster y datos de Entidades, e incorpora semántica de datos de observaciones. Además, un nuevo lenguaje declarativo combina constructores lógicos y funcionales que ya están presentes en otros lenguajes bien conocidos.

***Objetivo 3*:** Implementación eficiente del procesamiento espacio-temporal on-line declarativo en arquitecturas hardware distribuidas a gran escala.

A continuación, se detallan las principales contribuciones de esta Tesis:

– Generalización de un servidor de adquisición y diseminación de datos, con gran aplicación un muchos dominios científicos e industriales, que proporciona flexibilidad en la incorporación de diferentes tecnologías para la adquisición, almacenamiento y diseminación de datos.

– Definición de un novedoso paradigma híbrido lógico-funcional que permite la formalización de un modelo de datos para la gestión integrada de datos de entidades y datos muestreados.

– Definición de un nuevo lenguaje espacio-temporal declarativo de análisis de datos que aprovecha el modelo de datos mencionado anteriormente.

– Definición de un modelo de datos para Almacenes de Datos que proporciona semántica de datos de observaciones, incluida la aplicación del lenguaje antes citado en la definición declarativa de procesos de observación ejecutados durante la carga de datos de observaciones.

– Implementación distribuida, paralela y columnar del lenguaje declarativo de análisis espacial. La ingente cantidad de datos que deben ser procesados fuerza la utilización de las infraestructuras cluster multi-nodo y arquitecturas hardware multi-core que existen actualmente.

El diseño e implementación de GeoDADIS ha significado un esfuerzo de generalización hacia el desarrollo de servidores de adquisición y diseminación de datos. GeoDADIS propone una nueva arquitectura escalable y extensible que soluciona el problema de la heterogeneidad en el acceso y diseminación de datos de sensores. La arquitectura general de GeoDADIS está dividida en tres capas software. La funcionalidad que se encarga del control del sistema, así como de la gestión de los datos y metadatos de configuración, está proporcionada por los componentes de la capa Gestión de Datos y Control. Los procesos de muestreo de la capa Adquisición de Datos permiten la adquisición de medidas producidas por sensores heterogéneos a través de canales de adquisición externos. La capa Interacción Externa permite la interacción de GeoDADIS con usuarios, aplicaciones externas y administradores. Se permiten dos tipos de comunicación. La aproximación publicación/suscripción permite la suscripción a los datos de forma que sea GeoDADIS el que envía los datos a los suscritores a medida que los

va obteniendo. La aproximación cliente/servidor permite a los clientes consultar directamente los datos almacenados en el sistema. La flexibilidad se consigue en GeoDADIS gracias al uso de diferentes patrones de diseño software tanto en la implementación como en el diseño de sus diferentes componentes. El patrón *Adapter* facilita la incorporación de nuevos servicios de datos, servicios de control remoto y canales de adquisición de datos con cambios mínimos en los componentes internos del sistema. La incorporación de dichos elementos requiere únicamente de actualizaciones de la información de configuración. La flexibilidad, escalabilidad y extensibilidad han sido validadas durante el desarrollo de un prototipo para adquisición y diseminación de datos basado en GeoDADIS que permite la monitorización del estado de salud en entornos educativos.

El diseño de SODA se ha dividido en diferentes tareas. En primer lugar, se ha definido un modelo de datos espacio-temporal que incluye nuevos tipos (espaciales y temporales), y estructuras de datos (Dimensiones, Extensional MappingSets, Intensional Mappings) necesarias para la correcta representación de datos de Entidades y datos Raster de forma integrada. Sobre este primer modelo de datos se ha construido un modelo de datos que dota al sistema de la semántica de observación requerida gracias a la definición de un nuevo lenguaje llamado XODDL. A continuación, se ha definido un lenguaje declarativo espacio-temporal para el análisis de datos llamado MAPAL. Además de la especificación de datos y tareas de análisis, este lenguaje permite la definición de procesos analíticos. Estos procesos se ejecutan de forma interna y proporcionan nuevas observaciones a partir de observaciones externas registradas por los diferentes canales de adquisición. Finalmente, se han definido los operadores de sistema que se encargan de ejecutar las tareas definidas por el usuario en MAPAL. Las ventajas principales de SODA se puntualizan a continuación:

- Tanto el modelo de datos de observaciones espaciales como la definición declarativa de los procesos analíticos incorporan y dan soporte a la semántica de datos de observaciones.
- Se proporciona soporte directo a la representación y análisis integrado tanto de datos E/R convencionales como datos espaciales, temporales y espacio-temporales muestreados.
- Los nuevos tipos de datos espaciales y temporales permiten la representación y transformación entre diferentes resoluciones tanto en el dominio espacial como temporal.

– El concepto matemático de función se utiliza para representar tanto datos (mediante la nueva estructura de datos Extensional MappingSet) como comportamiento (mediante la nueva estructura de datos Intensional Mapping). Así pues, esta solución debería ser de fácil uso para usuarios del ámbito científico. Adicionalmente, esta aproximación funcional facilita la definición y reutilización de resultados intermedios.

– La incorporación de los nuevos lenguajes propuestos, MAPAL y XODDL, en servicios web es muy sencilla debido a que están basados en el lenguaje declarativo XML.

– La implementación eficiente de SODA se ha visto beneficiada por las estructuras de datos no anidadas que se han definido en el modelo de datos.

Se ha propuesto también la implementación de un prototipo de forma que se pueda comparar a SODA con las soluciones existentes en el estado del arte para el análisis de datos espaciales y espacio-temporales. Los beneficios de los modelos de datos y operadores definidos se han demostrado en los resultados de rendimiento obtenidos por el prototipo implementado. Los tiempos de ejecución obtenidos para la operación de join espacial, implementada para esta comparativa, mejoran los obtenidos por las soluciones existentes actualmente (p. ej., GeoSpark, LocationSpark, Stark). Dichos tiempos de ejecución son órdenes de magnitud inferiores a los obtenidos por los competidores. Además, los test de escalabilidad demuestran un rendimiento similar al de las mejores soluciones actuales.

El principal inconveniente de SODA viene dado por la adopción de un nuevo paradigma funcional de gestión de datos por parte de los usuarios de bases de datos tradicionales. Sin embargo, el formalismo funcional se ha combinado con el bien conocido formalismo lógico a la hora de definir MAPAL. Por tanto, los constructores de MAPAL son muy similares a los constructores que están presentes en otros lenguajes disponibles actualmente como XQuery.

Para finalizar este resumen, se van a comentar las líneas de trabajo futuro que se pueden derivar de esta Tesis.

Respecto a GeoDADIS, las futuras líneas de investigación deberían estar relacionadas con la ampliación del sistema de forma que se pueda dar soporte para la adquisición y diseminación de las observaciones complejas que producen los sensores remotos (p. ej., lidar, radar).

En cuanto a SODA, se pueden identificar diferentes vías de trabajo futuro. A continuación, se detallan las más relevantes:

- Incorporación de nuevas técnicas de optimización de consultas.
- Definición de nuevas estructuras de indexación.
- Diseño e implementación de nuevas estrategias de particionamiento para Dimensiones, Extensional MappingSets y datos espaciales.
- Incorporación de técnicas de procesamiento aproximado de consultas sobre Extensional MappingSets almacenados.

CHAPTER 1

INTRODUCTION

1.1 Background

A myriad of data acquisition devices is observing every day more variables and generating a vast amount of data in almost every application domain, e.g., health care, home automation, clean energy production, weather forecast, natural disaster prediction. Furthermore, an increasing number of research areas are involved in the efficient acquisition and management of data, e.g., Sensor Networks, Data Logging, Internet of Things (IoT), large scale data management, and in the intelligent exploitation of information, e.g., Data Analytics and Mining.

Environmental observation data is an essential portion of such generated data, whose spatio-temporal nature has posed interesting data management challenges. More specifically, important research efforts have been devoted to Environmental Observation Data Management Systems for decades. Two features are common to all these systems: *spatio-temporal observations* and *heterogeneity*.

Spatio-temporal observations

The location of each observation in some reference space and the time when the observed value of an observation applies to the observed property are important pieces of metadata that must be used during the analysis. In the context of this Thesis, the Observations and Measurements (O&M) conceptual schema [30] of the Open Geospatial Consortium (OGC) was adopted as the theoretical framework for the definition of the concept of *observation* and other related concepts (e.g., observed property, observed value). An *observation* en-

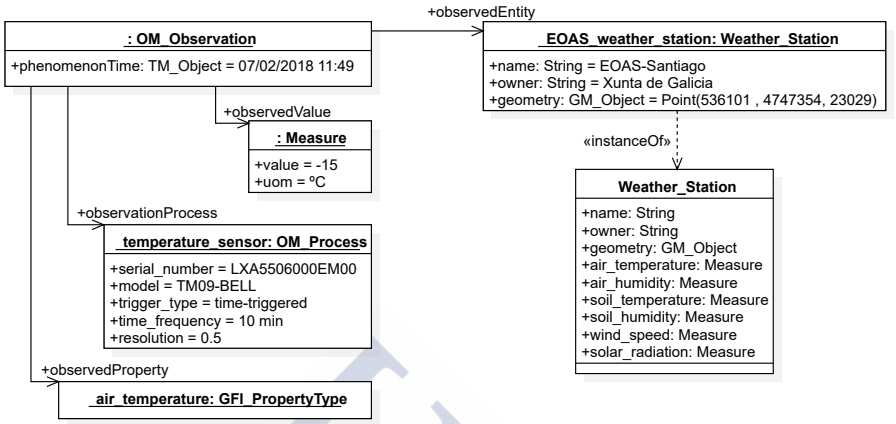


Figure 1.1: OGC Observation example.

closes both an *observed value* and the relevant observation metadata that provides observation semantics required to adequately interpret it. An example of an *observation* is shown in Fig. 1.1. An *observed value* (`-15`) with a specific *unit of measure* (`°C`) of an *observed property* (`air_temperature`) is provided by an *observed entity* (`EOAS_weather_station`). An *observed entity* may have both conventional properties (`name`, `owner`, `geometry`) and *observed properties* (`air_temperature`, `air_humidity`, `soil_temperature`, `soil_humidity`, `wind_speed`, `solar_radiation`). Values of conventional properties are usually assigned by some authority whereas values of *observed properties* are estimated by some *observation process* (`temperature_sensor`). It is mandatory to register properties (`serial_number`, `model`, `trigger_type`, `time_frequency`, `resolution`) of the specific *observation process* used to generate the *observed value*. *Observation processes* may be of very different nature, including physical devices (e.g., temperature sensors), tasks performed by people (e.g., data registered by an operator) and data processing algorithms (e.g., weather forecast). It is also mandatory to register the *phenomenonTime* (`07/02/2018 11:49`), i.e., the time instant when the *observed value* applies to the *observed property*. Notice for example that the weather forecast (observation process) might take into account historic data values obtained some time ago. The type of observation data produced by an *observation process* is determined by two characteristics: i) whether the process is executed periodically (`time_triggered`) or triggered by specific events (`event_triggered`); ii) the relative position of the process with

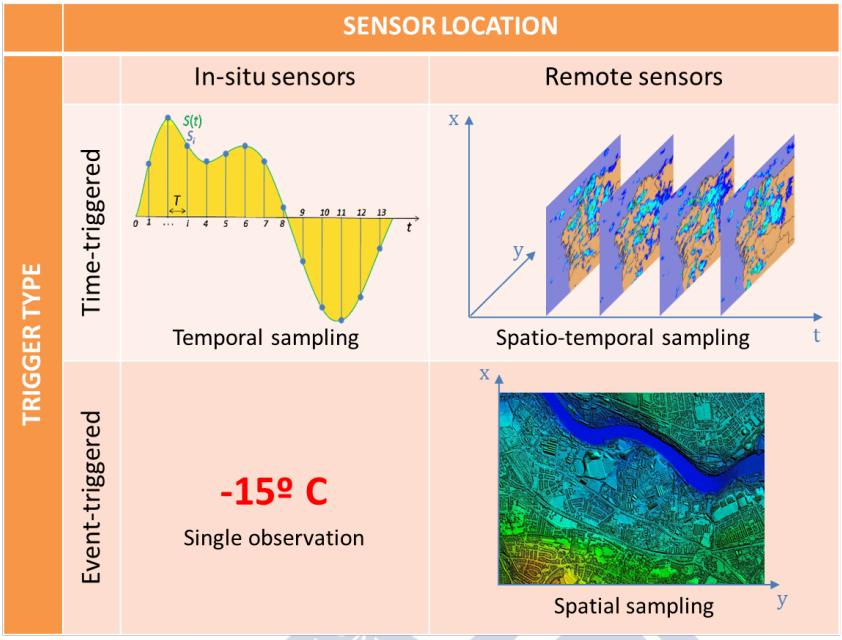


Figure 1.2: Observation data types.

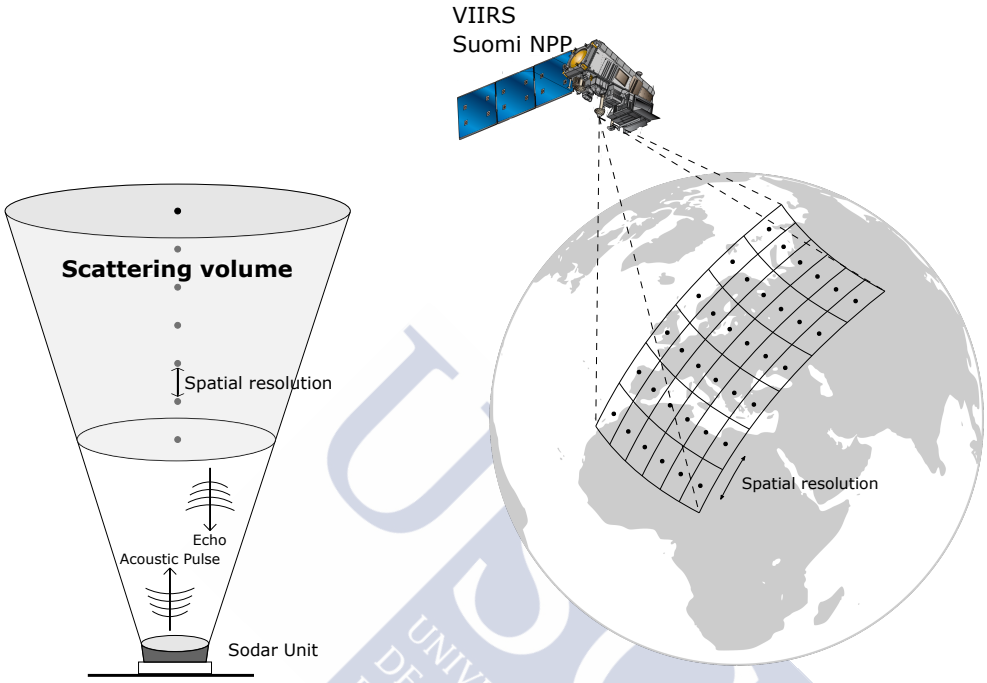
respect to the observed entity (`in-situ` or `remote`). Different available data types obtained by combining such characteristics are shown in Fig. 1.2.

Trigger type

Event-triggered processes start at some time instant determined by a specific event. For instance, a Light Detection and Ranging (LIDAR) image taken at some specific time instant. *Time-triggered* processes are executed at some predefined time frequency producing regular samplings in the temporal domain. As an example, we might register air temperature values obtained by the temperature sensor of a weather station every ten minutes.

Sensor location

Focusing on sensors (one of the aforementioned *observation process* types), *in-situ* sensors are located at the spatial position of the *observed entity*. They produce a single observa-



(a) SODAR (static platform) (b) VIIRS (mobile platform)

Figure 1.3: Illustration of 1D and 2D spatial samplings.

tion value at each time instant. Examples of such sensors are a temperature sensor installed in a meteorological station (static platform) and a GPS device installed in a car (mobile platform). Unlike *in-situ* sensors, *remote* sensors are located far away from the *observed entity*. They provide several *observed values* (one for each *observed entity*) at each time instant. An example of static remote sensor is the Sonic Detection And Ranging (SODAR), Fig. 1.3(a), used to register wind speed at different heights above the ground by measuring the scattering of sound waves produced by atmospheric turbulence. SODAR generates a 1D sampling of wind speed along consecutive discrete locations of a vertical line profile. An example of a remote sensor installed in a mobile platform is the Visible Infrared Imaging Radiometer Suite (VIIRS) installed in the Suomi NPP satellite (Fig. 1.3(b)). VIIRS allows high resolution images to be acquired both in visible and infrared spectrum, providing a whole view of the Earth every two days with a spatial resolution of 750 meters. Generated data include 2D

regular samplings (called *Rasters* in the area of Geographic Data Management) of color and temperature of the sea surface.

Heterogeneity

Heterogeneity specifically concerns the data acquisition part of the aforementioned systems, which need to access data produced by heterogeneous sensing devices (e.g., humidity sensors, GPS devices, radar, lidar, multispectral scanner sensors) following different software/hardware specifications that are accessed through several communication protocols (e.g., WiFi, RS-485, Ethernet).

General system architectures of relevant Data Acquisition and Monitoring Applications [62, 102, 76, 78, 63], and Non Real-Time Supervisory Control Systems [26, 32] are usually composed of three main components, namely, *End-User Applications*, *Data Servers* and *Sensing Devices*. *End-User applications* are in charge of data analysis and visualization. *Sensing devices* run the observation process and are highly heterogeneous both in functionality and communication capabilities, as already stated. Acting as a gateway between the heterogeneous specific domains of *End-User Applications* and the heterogeneous collection of *Sensing Devices*, one or more *Data Servers* are added to the system in order to provide homogeneous data access.

1.2 Problem description

Based on the above, several challenging problems arise during the design and implementation of Environmental Observation Data Acquisition and Management Systems. Specific issues related to both observation data acquisition and observation data management are detailed below.

Regarding observation data acquisition, generalization efforts in *sensing devices* programming and *end-user application* development tend to be worthless because of the strong conditioning on vendor specifications and the high dependency on specific domain and user preferences, respectively. On the other hand, the functionality and architecture of *data servers* tend to be very similar in the broad majority of applications. A major challenge however is to provide the required flexibility to enable data acquisition from heterogeneous *sensing devices* and data dissemination through heterogeneous *end-user applications*. The system must provide simple and straightforward mechanisms for the incorporation of the following components:

- New *in-situ sensing devices*.
- New data *dissemination services*.
- Different persistent data storage technologies for different *observed properties*.

Focusing on observation data management, a system must provide the following general functionalities to effectively manage observation data:

- Management of conventional Entity/Relationship (ER) data related to non-observed properties of *entities*.
- Management of sampled data over temporal, spatial (1D and 2D) and spatio-temporal domains.
- Support for observation data semantics. Relevant observation metadata of *observed properties* of *entities* must be provided.
- Efficient implementation for large scale shared-nothing distributed hardware architectures.

1.3 Motivation

In terms of spatial data software architectures for GIS, recent developments and trends propose the decomposition of systems into simple and well-defined services, which are often web-based and whose interfaces follow international interoperability standards of the OGC and the International Organization for Standardization (ISO). Thus, Spatial Data Infrastructures (SDI) integrated by distributed services through the Internet can be made available to GIS developers.

Beyond the previous technological consideration, relevant policies are being adopted to improve the availability of spatial data sets generated by different public administrations. In particular, the INSPIRE Directive of the European Union (2007/2/CE, March 14th 2007) encourages the creation of a SDI to ensure the interoperability of spatial information systems in Europe. The application of INSPIRE in the Spanish legislative system forces public administrations to make their geographic data available through SDI services. Therefore, the new enriched geographical knowledge allows for the appearance of many applications in different areas of knowledge that require spatial analysis capabilities.

In spite of the above needs, to the best of my knowledge, none of the available technologies and approaches found in data acquisition and data management literature provide support for all the functionalities required in Section 1.2. More details related to the this assertion are given in the following paragraphs.

Most of data acquisition systems provide high flexibility to obtain observation data from heterogeneous sensing devices but lack the required flexibility in data storage and dissemination capabilities. As opposed to [79], in [17, 26, 53, 67, 89, 96] flexible mechanisms to attach new sensing devices are provided. However, [96] lacks flexibility to extend implemented data storage technologies whereas [17, 26, 53, 67, 89] provide limited capabilities. Flexible ways to attach new dissemination services are provided in [79] as stored procedures and user-defined functions accessible through web-form interfaces. Such flexibility is not available in [89, 96] and is very limited in [17, 26, 53, 67].

A huge amount of research effort devoted to observation data management may be found in data management literature. The area of spatial databases [46, 68] is one of the most active research areas providing many research approaches. Even the ISO SQL standard [58], implemented by well known DBMSs [83] has been extended with relevant spatial functionality. These tools currently enable declarative querying over spatial data, including support for 2D rasters. High performance Data Warehouse [54] and NoSQL [77] tools implement spatial extensions although raster data are not supported. Recording and processing of conventional and spatial data, including rasters, are currently supported by available GIS tools [81]. Declarative data analysis, not provided by such GIS solutions, is supported by array data managers [15, 22] for very large collections of array raster data. Even though declarative analysis of relational data through array data structures is not very user friendly, an attempt of integrated management of relational and array data was tried in [111] but the user has to deal with both relational and array semantics. Systems providing declarative analysis of data streams of sensor data have been developed in [40, 71]. However, raster data are not supported. Finally, observation data semantics are only supported by standards of OGC, Sensor Web Enablement (SWE) initiative [30, 84, 23] and specific observation data models and ontologies [20, 29, 72], although declarative analysis of observation data is not supported.

1.4 Objective and contribution

Based on all the above characteristics of currently available solutions and approaches, the main objective of this Thesis is the design and implementation of a generic framework for spatio-temporal observation data acquisition and declarative analytical processing. This overall goal can be divided into three independent specific objectives.

Objective 1 **(GeoDADIS):** Design and implementation of a generic observation data acquisition and dissemination server. Generic functionality of this framework must enable the following features:

- *In-situ* observation data acquisition through both synchronous and asynchronous data channels.
- Observation data dissemination through both client/server and publish/subscribe data services.
- Simple and straightforward incorporation of new observation data storage technologies.

Objective 2 **(SODA):** Design of a framework for declarative spatio-temporal analysis in very large spatio-temporal data warehouses. The well known mathematical concept of *function* is the foundation of a new simple data model which integrates entity-based and raster data, and incorporates observation data semantics. A novel declarative language combines logical and functional constructors already present in other well know languages.

Objective 3**:** Efficient implementation of spatio-temporal on-line analytical processing in large scale distributed shared-nothing hardware architectures.

Based on the above objectives, the general system architecture shown in Fig. 1.4 is proposed in this Thesis.

The main contributions of this Thesis may be summarized as follows.

- Generalization of a data acquisition and dissemination server, with great applicability in many scientific and industrial domains, providing flexibility in the incorporation of different technologies for data acquisition, data persistence and data dissemination.

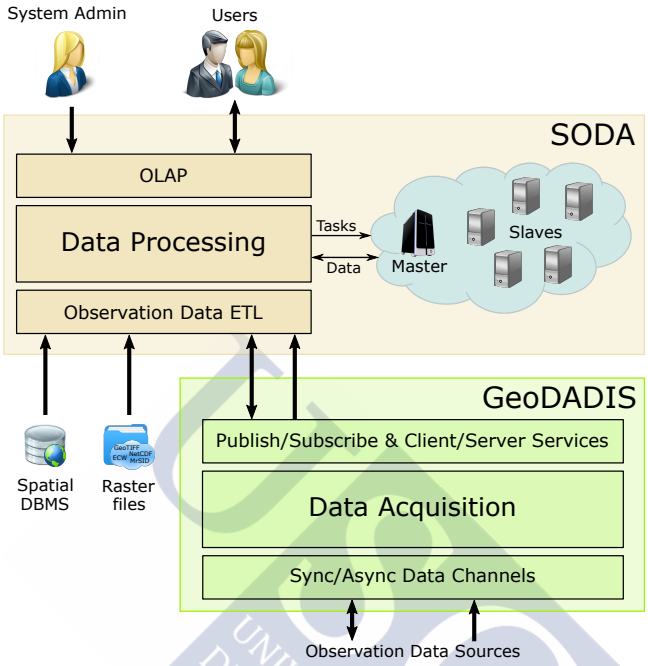


Figure 1.4: System Architecture.

– Definition of a new hybrid logical-functional paradigm to formalize a novel data model for the integrated management of entity and sampled data.

– Definition of a novel spatio-temporal declarative data analysis language for the previous data model.

– Definition of a data warehouse data model supporting observation data semantics, including application of the above language to the declarative definition of *observation processes* executed during observation data load.

– Column-oriented parallel and distributed implementation of the spatial analysis declarative language. The huge amount of data to be processed forces the exploitation of current multi-core hardware architectures and multi-node cluster infrastructures.

1.5 Outline of the Thesis

The Thesis dissertation has been organized as follows. Chapter 2 provides a detailed survey of related state of the art. First, current data acquisition solutions are introduced and compared with the solution proposed in this Thesis. Then, data analysis systems are reviewed and classified by their features. Two sections of this chapter are devoted to specific types of these systems, those designed for distributed spatial data processing and distributed spatio-temporal data processing respectively.

Chapter 3 provides a detailed description of GeoDADIS. First, the general architecture of GeoDADIS is introduced in Section 3.2. Next, the most important components of GeoDADIS architecture and specific elements enabling communication between them are explained in depth in Section 3.3. Finally, an experimental implementation of such architecture enabling the integration of environmental and health data is provided in Section 3.4.

In Chapter 4 the design of SODA is described. The underlying spatio-temporal and observation data models enabling the representation of spatio-temporal and observation data respectively are defined in Section 4.2. Then, an observation data analysis framework is defined in Section 4.3. In Section 4.3.1 a novel XML-based declarative language, called MAPAL, for the definition of analysis tasks is described. The use of such language to define analytical processes to be executed inside the framework is illustrated in Section 4.3.2. Then, system operators defined to execute analytical tasks provided by users are broadly explained in Section 4.3.3. Finally, in Section 4.3.4, an example of how MAPAL sentences are translated into system operators to be executed by the framework is exposed.

A prototype implementation of the spatio-temporal data model and the observation data analysis framework is broadly detailed in Chapter 5. Section 5.2 is devoted to the implementation of defined data types. Section 5.3 provides an extensive explanation of both in-memory and disk data structures implementation. User-defined data types and user-defined functions required to properly represent MAPAL data types and primitive mappings within Spark DataFrames are described in Section 5.4.1 and Section 5.4.2 respectively. An implementation of the flexible structure defined to add new data channels to the framework is covered in Section 5.5. Besides, two specific data channels defined to import and export GeoTIFF and PostGIS data have been implemented for illustration purposes. Section 5.6 details the implementation of system operators defined in Section 4.3.3. Finally, Section 5.7 shows the results obtained from the prototype evaluation. Current state of the art solutions are compared

to our prototype by testing the performance of the spatial join operation. Both execution time and scalability results are provided.

Chapter 6 highlights the major conclusions of this Thesis and proposes future lines of research.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Introduction

In this chapter a detailed description of related research work is discussed. For an easier understanding the discussion is divided here into four sections. First, several approaches related to the area of Data Acquisition Systems are exposed in Section 2.2. Next, state of art in the area of Data Analysis Systems is discussed in Section 2.3. Section 2.4 is devoted to related work in the area of Parallel and Distributed Proccessing of Spatial Data. Finally, approaches for Distributed Spatio-Temporal Data Processing are described in Section 2.5.

2.2 Data Acquisition Systems

There is a plethora of sensor networks that have been developed and widely deployed in almost every application domain as a product of the vast research effort carried out during the last three decades. Specifically, monitoring and control applications have been seen as main targets of such effort during the last decade.

As stated in [28], a Distributed Control System (DCS) is composed of heterogeneous collections of physical devices connected through different communication protocols. Although the full-control functionality of a DCS is out of the scope of the data acquisition capabilities of GeoDADIS (e.g., real-time and fault tolerance constraints), similarities between some DCS components and GeoDADIS may be found.

2.2.1 CORFU Framework

A Common Object-oriented Real-time Framework for the Unified (CORFU) development of distributed IPMCS (Industrial Process Measurement and Control Systems) applications is defined in [96]. The CORFU framework adopts the *function block* concept defined by IEC standards [55, 56] and proposes a new network topology for fieldbus interconnection. The core element in the proposed network topology, called *interworking unit*, is composed of the following building blocks.

- *Virtual Field Bus* (VFB): the main component of the *interworking unit* abstracts any commercial fieldbus to the IEC 61499 [55] level. This abstraction allows for interoperability in fieldbus level.
- *Fieldbus Wrapper*: allows for wrapping different fieldbus specifications to the VFB.
- *Industrial Process-Control Protocol* (IPCP): each *interworking unit* implements the IPCP on top of TCP/UDP layers. The IPCP has been defined for the development, distribution, and operation of function block based industrial process measurement and control applications.

In this solution the *interworking units* are located between each fielbus and a backbone network that provides real time interconnection of fieldbus segments. The architecture of the *interworking unit* adopts the Adapter pattern [41] to ease the incorporation of new wrappers that enable the interconnection of heterogeneous fieldbuses to the selected backbone. Similarly, GeoDADIS implements the Adapter pattern to access data acquisition channels, data services and control clients.

2.2.2 TORERO Project

The research project TORERO (total life cycle web-integrated control) specifies a new DCS environment. The main element of the TORERO DCS [89] is a mechatronic component, called *torero device*, providing intelligent control. In a TORERO environment the required control functionality is realized by all *torero devices* working in collaboration. The architecture of a *torero device* is divided into three layers:

- *Physical layer*: sensor/actuator elements.

– *Hardware layer*: processor, storage, RAM, Ethernet interface, connectors to the sensor/actuator elements, etc.

– *Software layer*: Operating System, Java Virtual Machine, FTP, HTTP, the control application, etc.

The control application can only access hardware components via so called *device functions*. A *device function* is an abstraction of the underlying hardware, i.e., a wrapper that allows heterogeneous hardware to be controlled by the same control application software. As stated in Subsection 2.2.1, GeoDADIS also implements the Adapter pattern.

2.2.3 Chimaris and Papadopoulos, 2007

A generic component-based framework that can be used to build telecontrol applications was defined and implemented in [26]. The main components of this framework, called *remote units*, are small intelligent subsystems. Each *remote unit* must perform the following tasks.

– Handle every connected device (alarms, lights, heating, etc.).

– Monitor the connected devices and transmit data changes and message alerts to the control center.

– Change its behavior based on received update and control messages.

– Support secure and consistent communication.

– Ensure the availability and efficiency of the communication channel.

Based on the above, a *remote unit* may include control functionality that goes beyond the control capabilities of GeoDADIS. However, the flexibility requirements imposed during the design of GeoDADIS for data dissemination, data storage and remote control are not present in [26].

2.2.4 Horsburgh et ál., 2011

An environmental observatory information system that supports collection, organization, storage, analysis and publication of hydrologic observations is described in [53]. The architectural and procedural components are described as follows.

– *Data Observation and Communication*: sensors and telemetry systems used to collect observations.

– *Data Storage and Metadata*: data models, database systems and software required to create a persistent data repository.

– *Quality Assurance, Quality Control and Provenance*: software and procedures for transforming raw data into publishable data products.

– *Data Publication and Interoperability*: software, protocols, formats and vocabularies used for publishing data in interoperable formats.

– *Discovery and Presentation*: tools provided to data consumers for visualization and analysis purposes.

Related to GeoDADIS are the *Data Storage and Metadata* and *Data Publication and Interoperability* components. The former enables persistent storage of both sensor data and metadata. An important added-value step in this component involves the mediation across the variety of software supporting sensor and communication systems. Such a mediation is achieved in GeoDADIS by the implementation of wrappers for different data acquisition channels, as already mentioned. The latter provides data dissemination functionality, achieved in GeoDADIS by the implementation of data services.

2.2.5 GEOSWIFT Infrastructure

GeoSWIFT is a distributed geospatial infrastructure for the Sensor Web[1] proposed in [67]. The core component of GeoSWIFT is the open geospatial *sensing service*, which serve as a single queryable global sensor for Sensor Web users. Each *sensing service* role and its behavior are explained below.

– *Sensing Server*: provides a web-enabled interface for sensor systems and their geospatial information. The standard for sensor data access exposed by GeoSWIFT is based on the specifications provided by the Sensor Web Enablement (SWE) initiative [19] of the OGC.

[1] "A Sensor Web is a system of intra-communicating spatially distributed sensor pods that can be deployed to monitor and explore new environments" [61].

– *Sensing Registry*: plays a central role in publishing, finding, and binding to network-accessible services by providing a common mechanism to classify, register, describe, search, maintain, and access information about Sensor Webs and other Web Services.

– *Viewer*: GeoSWIFT Viewer is based on GeoServNet Viewer[2].

As in the case of GeoDADIS, the *Sensing Server* of GeoSWIFT acts as a gateway that hides the different communication protocols, data formats and standards of sensor systems and provides a standard interface for clients to collect and access sensor observations. Despite of the similarities between GeoSWIFT and GeoDADIS, GeoSWIFT does not achieve the flexibility requirements imposed during the design of GeoDADIS.

2.2.6 LIFE UNDER YOUR FEET (LUYF) Sensor Network

A data access gateway is implemented in [79] to gather data from a Wireless Sensor Network (WSN) for soil monitoring. Core components of LUYF are detailed below.

– *Data Collection Subsystem*: composed of motes and a base station. Each mote is connected to a data acquisition board providing ambient light, temperature and soil moisture sensors. Motes sample data at some predefined temporal resolution, typically every minute, and store them in local memory. The base station requests stored data from motes once every two weeks and stores the retrieved measurements in the database.

– *Database*: raw measurements arrive from the base station as ASCII files. First, received data are loaded into a temporary table. Next, duplicates are removed and data are stored as raw data. A multi-step pipeline is required for converting raw data to scientifically meaningful values. Such process is automatically performed by a stored procedure for all sensors within the database. Stored procedures and user defined functions, accessible through web-form interfaces, provide access to aggregated data.

The major drawback of LUYF, when compared to GeoDADIS, is the lack of flexibility that allows users to add new data acquisition wrappers.

[2] GeoServNet Viewer is a 2D/3D Web GIService viewer designed for streaming large amount of spatial data via Internet.

2.2.7 SPINE Framework

The general architecture of the SPINE framework [17] is composed of a collection of *sensor nodes* connected to the *coordinator node* that manages the network, collects and analyzes the retrieved data, and acts as a gateway to connect sensors and wide area networks. The *sensor node* manages and abstracts sensors (providing a standard interface to diverse sensor drivers), and is responsible for sampling and storing sensor data in properly defined buffers. Two major differences may be found between SPINE and GeoDADIS. First, SPINE enables the incorporation of signal processing functionality that is out of the scope of GeoDADIS. Second, the flexibility in the incorporation of new data dissemination and remote control services of GeoDADIS is not present in SPINE.

2.3 Data Analysis Systems

In this section a comparison between different solutions in the area of data analysis is provided. Based on generic functionalities required for all observation management systems, the comparison criteria are specified below.

1. *Direct support for observation semantics*[3]: the representation of terms related to an *observation* is required. *Observed entities* allow for the representation of *entities* with conventional and *observation properties*. Effective analysis and correct interpretation of *observed values* of some *observed property* require relevant metadata to be recorded. Specifically, important metadata to be recorded are the *observation process* and the *phenomenon time*. *Observation process* and *observation entity* instances have to be classified into *process types* and *entity types* respectively. Moreover, the recording of *observation process* properties should be also supported.

2. *Support for the management of sampled data*: it is not only for classical E/R data that an observation data management system must support efficient processing. Data structures and operations have to be provided to enable effective processing of sampled data. As aforementioned in Section 1.1, *time-triggered processes* generate temporal sampling data and remote sensors usually produce spatial sampling data (raster data). As we will see below, either highly inefficient approaches or complex nested models arise when applying the classical relational-based models to sampled data.

[3] We refer here to observation semantics provided by [30] and detailed in Section 1.1

3. *Support for multi-resolution temporal and spatial data*: *observation* data is currently generated with different temporal and spatial resolutions by a huge amount of available sensors. Because of that, evaluation of operations often implies transformations between diverse temporal and spatial resolutions. An appropriate data type system should be provided by observation data management systems in order to simplify these transformations.

4. *Simple data modeling approach*: for evaluation purposes in the context of this Thesis, we are considering as non-simple data models those that fulfill one or two of the following features:

   – more than one non-nested data structure.
   – nested data structures including records and collections.

   It is clear that simple data models have some advantages over non-simple ones, e.g., an efficient implementation of a simple model is far more straightforward than a nested data model implementation, and implementation of different semantics in diverse data structures often results in not user friendly interfaces.

5. *Model based on a well known paradigm*: a rapid progression up the learning curve is enabled when data models are defined based on well known paradigms on account of many years of user experience.

6. *Stream processing approach*: stream processing approaches are required when real time prerequisites are present and there is not a large amount of data to be recorded. These systems implement small stored data structures and efficiently process input data streams in order to produce output data streams. Stream processing approaches are commonly known as Complex Event Processing[4] (CEP) and they rely on the evaluation of Continuous Query Language (CQL) expressions [13, 59].

7. *On Line Transaction Processing (OLTP) approach*: OLTP approaches are required when real time prerequisites are present with simple temporal patterns and there is a large amount of data to be recorded. This approach is traditionally supported by conventional DBMSs for reasonably large data collections and provided by both NoSQL [77, 8] and NewSQL [105] solutions in the new era of Big Data Management.

[4] Also known as Information Flow Processing Systems [31]

8. *On Line Analytical Processing (OLAP) approach*: OLAP approaches are required when real time prerequisites are not present and there is a large amount of data to be recorded. We usually identify these systems in Data Warehouse solutions implemented by Bussines Intelligence (BI) applications. Examples of high performance implementations are Hewlett-Packard Vertica [99], which is an evolution of C-Store [93], and the open source MonetDB database [54]. Recent research solutions on column-oriented technologies serve as a basis for efficient implementations of those Big Data solutions. Indeed, the column-based storage of relational data, instead of the traditional row-based storage, is the major contribution of these approaches. Main features are:

   – efficient compression techniques.
   – processing over compressed data.
   – columns not involved in computations are not retrieved from storage.

   As major drawback we can mention the inefficient performance of insertions, updates and deletions of data. This makes them suitable for data warehouses.

9. *Support for declarative processing*: taking into account that procedural solutions are dominant in application domains such as environmental applications handling sampled observation data, and that procedural approaches have well known disadvantages compared to declarative languages, it is clear the motivation for applying declarative data management technologies to these environmental applications.

10. *Support for aggregated queries*: effective observation data analysis in OLAP systems cannot be accomplished without statistical methods providing aggregation functionality.

11. *Support for iterative processing*: recursive queries are required in only few data management applications. This is the reason why such functionalities were out of the scope of first SQL implementations. Current ISO SQL standard and DBMSs vendors support a kind of limited recursion. Regarding the analysis of observation data in environmental applications, such functionalities are commonly required to perform many simulations. Examples of these are forest fire propagation, oil spills, and flooding. Thus, although it is not a key functionality, support for iterative processing is a desirable feature.

12. *Data processing based on a well known language*: as stated for data models, a clear advantage for data management systems is that the definition of query languages is based on well known paradigms.

13. *Distributed processing of spatial, temporal and spatio-temporal data*: traditional technologies are no longer suitable for processing the large amount of spatial, temporal and spatio-temporal data currently generated. In fact there is a growing demand for solutions that support high performance queries on such data. This makes distributed and parallel processing of spatial, temporal and spatio-temporal data no longer desirable but required. Hence, data management systems currently created have as essential requirement such cluster-based processing.

14. *Availability of efficient implementation*: a data management approach is really useful if it can be efficiently implemented. A prototype implementation demonstrates the approach feasibility and its use in real application domains shows its maturity.

The degree of compliance with the previous evaluation criteria is now detailed for several related research solutions and available technologies, including also the SODA framework. Table 2.1 provides an overview of such evaluation. For each approach, *P* represents that the relevant criterion is partially supported and *Y* represents that is completely supported. A more detailed discussion is given below.

2.3.1 OGC SWE Standards

The Sensor Web Enablement (SWE) of the OGC provides standards for interfaces of web services that are related to the management of environmental observation data. In particular, the Observations and Measurements (O&M) [30] and Sensor Model Language (SensorML) [84] were already mentioned in Chapter 1. The Sensor Observation Service (SOS) [23] defines a web service interface to query observation data collections, either stored or directly obtained from devices. Data is transferred between client and server in standard XML encodings of O&M and SensorML models. Query capabilities of SOS are limited to just filtering. Regarding data processing, OGC defines the Web Processing Service (WPS)[88] interface that enables the invocation of data processing algorithms through the web. Various implementations of the above standards already exist in the market, both with commercial and open source licenses. In general it is obvious that O&M provides appropriate support for the modeling of

	Obs. Semantics	Sampled Data	Multiresolution	Simple Model	Well Known Model	Stream Proc.	OLTP	OLAP	Declarative Proc.	Aggregation	Iterative Proc.	Well Known Lang.	Impl. Available
OGC SWE Stds.	*Y*	*Y*			*Y*								*Y*
Obs. Data Models	*Y*			*Y*	*Y*								
GIS		*Y*		*Y*	*Y*								*Y*
Sensor Stream				*Y*	*Y*	*Y*			*P*	*P*		*P*	*Y*
Spat. and ST DBMSs		*Y*			*Y*		*Y*	*Y*	*P*	*P*	*P*	*P*	*Y*
Spatial NoSQL							*Y*		*Y*				*Y*
Spatial HP DW				*Y*	*Y*			*Y*	*P*	*P*		*P*	*Y*
Array Data Managers		*Y*		*Y*				*Y*	*Y*	*Y*			*Y*
SciQL		*Y*			*P*			*Y*	*Y*	*Y*		*P*	*Y*
Dist. Proc. Systems				*Y*	*Y*				*Y*	*Y*		*Y*	*Y*
SODA	*Y*	*Y*	*Y*	*Y*				*Y*	*Y*	*Y*			

Table 2.1: Comparison of related technologies.

observation semantics and sampled data. Different spatial and temporal resolutions are supported but transformations are a user matter. The underlying object oriented data modeling approach with XML encodings is well known. However, nested structures are required to support sampled data. Declarative data processing is not supported at all as WPS just provides means for remote procedure calls.

2.3.2 Observation Data Models

Beyond the above O&M OGC standards, several data models and ontologies have been proposed to support observation data semantics [20, 29, 72]. They are based on well known paradigms and provide observation data semantics with simple data modeling approaches.

However, sampled data and multi-resolution is out of the scope of these models as well as any kind of data processing.

2.3.3 Geographic Information Systems (GIS)

Currently, a wide variety of GIS tools, both with commercial and open source licenses, are available. A representative example of them is GRASS [81], which supports the management of any kind of geographic data, including rasters, recorded in many different well known models and formats. Raster data management is usually formalized with relevant raster algebras [24]. Observation semantics are not considered in GIS and although the managed data may have many different spatial resolutions, transformations between them have to be explicitly done by the user to perform operations. Spatial data processing is a strength of tools like GRASS. However, it is performed by the execution of a very large amount of different commands. Therefore, a declarative language is missing. Notice that the user must know which is the functionality of each command and how to combine them, thus only expert users may take real advantage of spatial data analysis with GIS tools.

2.3.4 Sensor Stream Processing Approaches

Various Stream Processing approaches have been explicitly proposed for the management of data generated by sensor networks [40, 71]. Although they were defined for sensor data management, observation data semantics are not explicitly incorporated and are delegated to user interpretation. Any kind of spatial data management is out of the scope of these approaches. They support declarative real-time processing of streams with aggregation functionality based on SQL like languages. Real-time requirements of these approaches are clearly in conflict with the support of iterative processing.

2.3.5 Spatial and Spatio-Temporal DBMSs

Many temporal extensions have been proposed for the classical relational model [33, 91]. Recently, some characteristics have been incorporated into ISO SQL standard [64]. Various spatial [46, 68, 98] and spatio-temporal [47, 104] extensions to classical models have been proposed in the literature. Spatial functionality has already been added to ISO SQL standard [58], which is currently implemented by most of the available DBMSs (see [83] for an example). Direct support of observation semantics is out of the scope of spatial DBMSs. They

support management of conventional E/R data with a well known object-relational paradigm and SQL, where properties of entities might have spatial data types (`point`, `line`, `surface`, etc.). Extensions for raster data are also supported by some approaches and systems [83], however, they require nested structures and do not provide explicit support for multi-resolution. They can be used both for OLTP and OLAP, but in the general case they were not designed with Big Data requirements in mind. Regarding declarative data processing, it is only efficiently supported for non-sampled data and it includes both aggregations and SQL recursion for iterative queries. To manipulate raster data with SQL constructors it has to be unnested from a complex value of a raster data type, which is a highly inefficient task.

2.3.6 Spatial NoSQL Databases

Systems following a NoSQL approach and providing spatial data management capabilities are still few. An example is the extension of MongoDB [77] with support for the management of GeoJSON encoded data. Their functionality is very limited both in data modeling and processing.

2.3.7 Spatial High Performance Data Warehouses Approaches

As stated in Criterion 8, a great research effort has been recently devoted to column-based implementations for OLAP. But, according to the best of my knowledge, spatial functionality over a high performance implementation is only provided by MonetDB DBMS [54]. Main drawbacks are the following.

- Support for iterative processing, sampled data and observation semantics is not provided.
- Support for declarative processing is partially supported.
- Recursive queries are not implemented.

2.3.8 Array Data Managers

Current array data manager developments [15, 22] heavily rely on array algebras [16]. The core idea behind these array data managers is to provide high performance OLAP over very

large arrays by using a simple array data model. Moreover, aggregation functionality is provided by implemented declarative array query languages. Main drawbacks of these systems are detailed below.

– Support for iterative processing, multi-resolution and observation semantics is not provided.

– Implemented languages are quite cumbersome for DBMS users because of the array semantics and complex array operators, even when their flavors are very similar to the SQL one.

2.3.9 SciQL

SciQL [111] is a query language based on SQL intended to be used in science applications. Main features of SciQL are as follows.

– Support for *entities* and sampled data.

– Integrated analysis of array and relational data.

– Declarative queries with aggregation.

The current implementation of SciQL makes use of MonetDB [54] technologies and shows the following drawbacks.

– Recursion is not included for iterative processing.

– The relational model is more complex because of the inclusion of an array data structure.

– The array semantics added to SQL makes SciQL not very friendly to DBMS users.

– Support for multi-resolution and observation semantics is not provided.

2.3.10 Distributed Processing Frameworks

A strong research effort is being carried out in the area of data management devoted to the design and implementation of distributed analysis frameworks for large scale spatial and spatio-temporal data . Detailed descriptions of several leading approaches are provided in Sections

2.4 and 2.5, focused on spatial and spatio-temporal data respectively. Main shortcomings of these systems are detailed below.

- Support for stream and iterative processing is not provided.
- Some approaches provide declarative processing by defining novel declarative languages mainly based on SQL.
- Observation semantics functionality is out of the scope of these solutions.
- Only GeoTrellis [43] enables multi-resolution and sampled data analysis.

On the other hand, all of them provide simple data models and are based on well known languages and data models.

2.3.11 SODA

The main features of the SODA framework are now itemized in order to carry out a valid comparison in terms of a qualitative evaluation.

- Support for sampled data, multi-resolution and observation semantics is provided.
- The mathematical concept of function serves as a basis for its non-nested data structure.
- Declarative spatio-temporal analysis and aggregation functionality is incorporated.
- Functional and logical constructors are combined in the declarative language.
- The relational paradigm has not served as a basis for the definition of the data model or the query language.
- A distributed spatio-temporal observation data processing implementation is provided.

2.4 Distributed Spatial Data Processing Systems

The relevance of Spatial Computing [90] is increasing in recent years due to the explosion experienced in the volume of spatial data produced by mobile applications and devices. Numerous specialized spatial data processing systems have been developed in the last five years to grapple with the huge amount of spatial data currently generated. Some spatial analytics

systems (e.g., SpatialHadoop [37] and Hadoop GIS [5]) follow a disk-based strategy, i.e., they have been optimized for IO efficiency rather than memory usage. These systems do not have to worry about fault tolerance or computation distribution thanks to the underlying MapReduce [34] architecture. They usually provide a set of spatial operators that allows users to execute spatial queries outperforming spatial extensions of relational database systems by orders of magnitude [5]. However, these systems have two major drawbacks.

– Despite data reuse is very common in spatial analytics, these systems do not reuse intermediate data. Developed on top of Hadoop [9], intermediate results have to be written to Hadoop Distributed File System (HDFS) [50] preventing further data analysis.

– Since they have not been optimized for distributed memory usage, latency increases and throughput decreases when scaling to large scale spatial data.

In order to tackle the above challenges, Spark [109] and Spark-related systems (e.g., Spark SQL [14], Spark Streaming [110], GraphX [44], and MLlib [75]) implement *in-memory* computing over a cluster of commodity machines. A natural choice is to develop efficient and novel spatial processing systems based on Spark. Hence, several spatial data management systems have been implemented on top of the Spark architecture, e.g., GeoSpark [108], SpatialSpark [107], GeoTrellis [43], Magellan [73], LocationSpark [95], and Simba [106]. A detailed description of these systems is given in the following subsections.

2.4.1 Hadoop GIS

Hadoop GIS [5] is a scalable and high performance spatial data warehousing system for running large scale spatial queries on Hadoop. The main objective is to provide a scalable, efficient and expressive spatial query system that allows for the execution of analytical queries on large scale spatial data. The following are the main features of Hadoop GIS.

– Parallelization of multiple spatial query types and mapping of query pipelines into MapReduce by performing tasks such as spatial partitioning, implicit and parallel execution of spatial queries on MapReduce, and execution of effective methods to amend query results by handling *boundary objects*[5].

[5] A boundary object is an object whose spatial extent crosses multiple tile boundaries.

Algorithm 2.1 Hadoop GIS algorithm for spatial query processing.

```
1: Data/space partitioning
2: Storage of partitioned data on HDFS
3: Pre-query processing (optional)
4: for tile in input_collection do
5:     Index building for objects in the tile
6:     Tile based spatial querying processing
7: end for
8: Boundary object handling
9: Post-query processing (optional)
10: Data aggregation
11: Result storage on HDFS
```

source: *Aji et ál., in Proc. VLDB Endow., 2013* [5].

- Support for fundamental spatial queries such as *point*, *containment*, *join*, and complex queries such as *spatial cross-matching* (large scale spatial join) and *nearest neighbor queries*.
- Implementation of global partition indexing and local on demand spatial indexing.
- Support for spatial declarative queries. Integration with Hive [97] allows for the definition of an expressive spatial query language (extending HiveQL [52]), and for the automation of translation and execution of spatial queries.

Algorithm 2.1 shows the steps for running a typical spatial query in the MapReduce environment proposed by Hadoop GIS. Data partitioning is performed by Steps 1 and 2. Data tiles are generated by space partitioning in Step 1. Then, Step 2 assigns tile unique IDs (UIDs) to spatial objects, merges these objects and stores them into HDFS. Hadoop GIS takes into account two major issues when partitioning data. First, high *data skew*[6] may be present in the spatial dataset [80] causing high density partitioned tiles. To avoid this scenario, Hadoop GIS splits high density tiles into smaller ones and performs recursive data partitioning. Second, *boundary intersecting objects* have to be properly handled. The multiple assignment based approach is taken by Hadoop GIS to replicate and assign intersecting objects to each

[6] In a distributed computing context data skew occurs when data is unevenly distributed across partitions in the cluster. Under these conditions, computational load is unbalaced among workers and may lead to longer job execution times and lower cluster throughput.

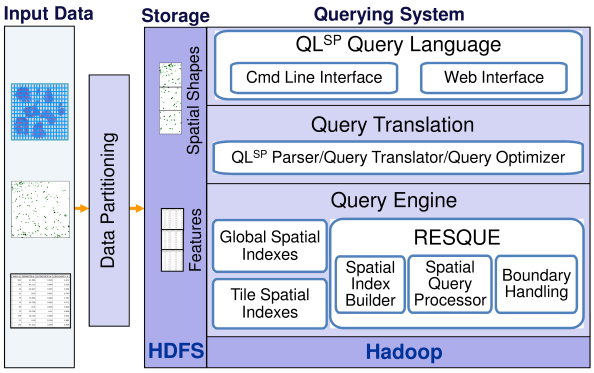


Figure 2.1: Architecture of Hadoop GIS.
source: *Aji et ál., in Proc. VLDB Endow., 2013* [5].

intersecting tile. The optional Step 3 is for pre-processing some specific queries, e.g., those that perform spatial data filtering based on global index. Parallel execution of spatial queries following the MapReduce paradigm is performed in Step 4. The main component to support spatial query processing in Hadoop GIS is the REal-time Spatial QUery Engine (RESQUE), as shown in Fig. 2.1. Two different spatial access methods for query processing are implemented in Hadoop GIS. A global region spatial index of partitioned tiles is used to filter HDFS file splits, and a tile based spatial index is used for spatial query processing. Identification of spatial objects within tiles is performed by Hadoop GIS in the mapping phase by using the tile name or UID to form the key for MapReduce. Depending on the query complexity, different spatial query pipelines (implemented as map functions, reduce functions or combination of both) can be executed in MapReduce. Step 8 is for remedying query results when handling boundary objects. As already stated, Hadoop GIS implements the multiple assignment approach which is simple to implement and fits nicely with MapReduce model. An optional post-query processing may be performed in Step 9. Data aggregation and storage of final results into HDFS are performed in Steps 10 and 11 respectively.

Hadoop GIS provides an integrated query language (QL^{SP}) over MapReduce as shown in Fig. 2.1. QL^{SP} extends HiveQL with spatial constructs, and spatial query translation and execution. Moreover, Hive query engine is extended with the RESQUE query engine ($Hive^{SP}$).

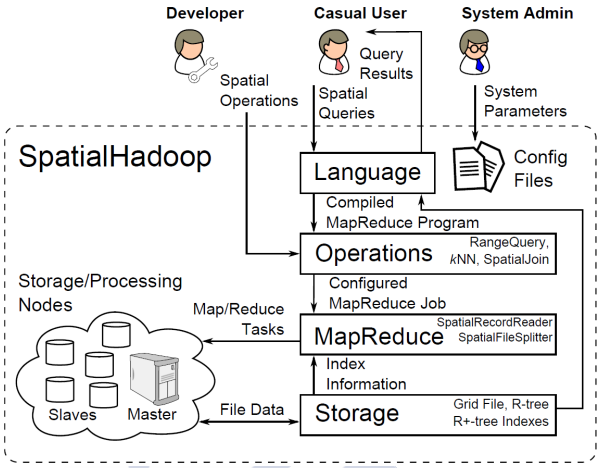


Figure 2.2: Architecture of SpatialHadoop.
source: *Eldawy and Mokbel, in ICDE, 2015* [37].

2.4.2 SpatialHadoop

SpatialHadoop [37] is a MapReduce framework with native support for spatial data. Unlike previous approaches (e.g., Parallel-Secondo [69], $\mathscr{MD}$-HBase [82], Hadoop GIS), SpatialHadoop do not rely on Hadoop as a black box. Such a different approach prevents SpatialHadoop from suffering the limitations and performance bottlenecks of Hadoop. The main attributes that allow SpatialHadoop to overcome the limitations of previous approaches are detailed below.

– Provision of *built-in* code. SpatialHadoop code is built inside the Hadoop base code to extend Hadoop core with spatial data functionality. This feature allows SpatialHadoop to be more powerful and efficient than previous solutions.

– Support for *skewed spatial data* distributions by implementing a set of spatial index structures.

– Users are enabled to develop a huge amount of spatial functions, e.g, *spatial join*, *range queries*.

As shown in Fig. 2.2, the SpatialHadoop architecture is divided into four main layers. A detailed description of each layer is provided below.

Language Layer. A novel high level SQL-based language, called Pigeon [36], is implemented in this layer. Several languages have been recently defined to reduce coding effort when working with MapReduce-based paradigms, e.g., HiveQL [52], Pig Latin [85], SCOPE [112], and YSmart [65]. Pigeon is an extension to Pig Latin providing OGC-compliant spatial data types, functions and operations. Standard spatial data types (e.g., `Point`, `LineString`, and `Polygon`) are supported. *User-defined functions* (UDFs) are harnessed to define spatial aggregations (e.g., `Union`), spatial predicates (e.g., `Overlaps`), and other spatial functions (e.g., `Buffer`). A new *k*NN (*k* nearest neighbors) statement has been added to support *k*NN-queries. In addition, two Pig Latin statements have been overridden. SpatialHadoop overrides the `Filter` statement to support range queries, and the `Join` statement to support spatial joins.

Storage Layer. As pointed out in [37], several challenges arise when applying traditional spatial indexes (e.g., Grid file, R-tree [48]) in Hadoop. To overcome these limitations, SpatialHadoop follows a two-layer indexing approach. A global index, stored in the master node, allows SpatialHadoop to split data across a set of partitions stored in slave nodes. A local index, stored in each partition, enables local data to be arranged. Regardless of the underlying spatial index structure, SpatialHadoop defines an index building algorithm composed of three main phases.

1. *Partitioning*. Spatial partitioning of the input file into *n* partitions performed through the following steps.

   a) Compute the number of partitions, *n*.

   b) Define partition boundaries, i.e., the spatial area covered by each single partition. This process strongly depends on the underlying index being constructed.

   c) Perform the physical partition of the input file, given the above partition boundaries, through a MapReduce job.

2. *Local Indexing*. A reduce function is used to build a local index on the stored data of each physical partition. To make this happen, the reduce function stores the records of each partition in a spatial index, written in a local index file.

3. *Global Indexing*. Once local indexing is performed, the master node builds a global index that indexes all partitions. First, concatenates all local index files into one final

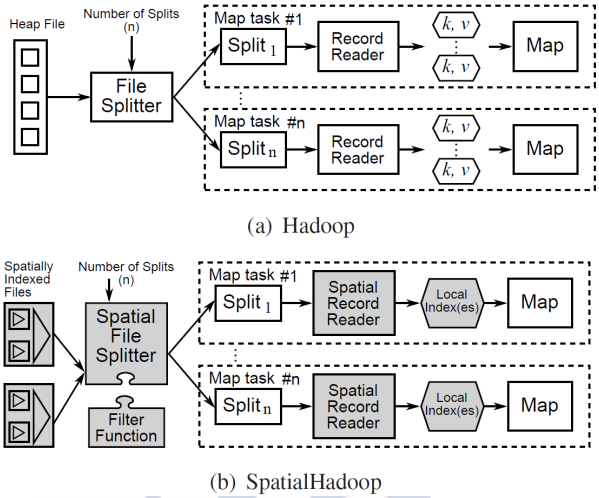


Figure 2.3: Map phase in Hadoop and SpatialHadoop.
source: *Eldawy and Mokbel, in ICDE, 2015* [37].

indexed file. Second, indexes all file blocks using their rectangular boundaries as the index key to build the *in-memory* global index.

MapReduce Layer. This layer is responsible for running the MapReduce jobs that process the required queries. Fig. 2.3 shows the *Map* phase of the MapReduce plan in both Hadoop and SpatialHadoop, highlighting the differences between them. In Hadoop, the *FileSplitter* takes the input file and divides the data into *n* splits, where *n* is determined based on the number of available slave nodes. Then, the *RecordReader* extracts records as *key-value* pairs and passes them to the *Map* function. SpatialHadoop enriches traditional Hadoop systems modifying the *FileSplitter* and *RecordReader* components. The new *SpatialFileSplitter* early prunes file blocks not contributing to the answer and generates data splits by exploiting the global spatial index stored on input files. And the new *SpatialRecordReader* efficiently process the previous splits using local indexes.

Operations Layer. The language layer is provided with a myriad of spatial operations (e.g., *range queries*, *kNN-queries*, *spatial joins*). The operations layer is responsible for the efficient implementation of all these spatial operations.

2.4.3 SpatialSpark

SpatialSpark [107] is a prototype system to process large-scale spatial join queries over Spark, and supports indexed spatial joins based on *point-in-polygon* test and *point-to-polyline* distance computation. The following main goals have been defined for SpatialSpark.

- Identify limitations and advantages of Spark for spatial data processing in cluster environments from an architectural point of view.
- Determine the potential performance of modern hardware for large-scale spatial join query processing.

Different indexing techniques for spatial filtering have been implemented in SpatialSpark. For spatial refinement, SpatialSpark relies on the well known Java Topology Suite (JTS) package [60]. To make SpatialSpark compatible with Hadoop-based systems, strings are used to represent geometries. Although higher efficiency could be possible by representing geometries as binary, avoiding string pairing overheads and allowing flexible disk accesses, this option is left for future work in SpatialSpark. As all intermediate data are memory resident in Spark, higher performance is achieved in SpatialSpark by minimizing expensive disk I/Os, and utilizing finer grained data parallelism.

2.4.4 GeoSpark

GeoSpark [108] is an *in-memory* cluster computing framework for processing large-scale spatial data, providing support for spatial data types, indexes, and operations by extending the core of Spark. Specifically, GeoSpark enhances the resilient distributed datasets (RDDs) to support spatial data (SRDDs). The key features of GeoSpark are the following.

- Support for loading, processing, and analyzing large-scale spatial data over Spark.
- Support for geometrical and distance operations is given by the definition of a set of SRDD types, e.g., `PointRDD` and `PolygonRDD`. Moreover, Spark programmers may easily develop spatial analysis applications by using the Application Programming Interface (API) provided by SRDDs.
- Support for different global spatial data indexing techniques. Input SRDDs are partitioned using a grid structure. Then, these resulting grids are assigned to computing machines for parallel execution.

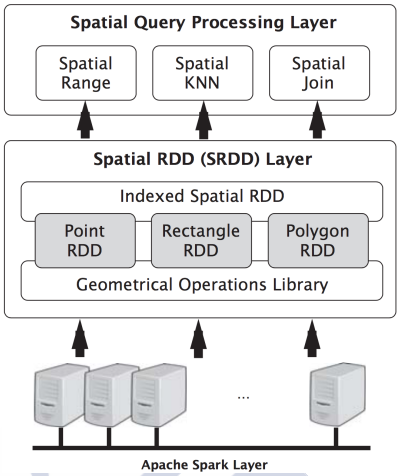


Figure 2.4: Architecture of GeoSpark.
source: *Yu et ál., in Proc. SIGSPATIAL, 2015* [108].

The architecture of GeoSpark is composed of three layers, as depicted in Fig. 2.4. *Apache Spark Layer* serves as the basis where GeoSpark is built on, *Spatial RDD Layer* provides support for geometrical and spatial objects and operations, and *Spatial Query Processing Layer* executes efficient spatial query processing algorithms.

Apache Spark Layer. Comprises the basic functions natively provided by Spark such as loading/storing data from/to persistent storage and regular RDD operations.

Spatial RDD Layer. Efficient partition of spatial data elements across cluster nodes is enabled by the definition of the extended spatial version of the Spark RDD. To write spatial data analytics applications, novel parallelized spatial transformations and actions in SRDDs provide users with an intuitive interface. Main features of this layer are pointed out below.

– *Spatial Objects*. Three new SRDDs (`PointRDD`, `RectangleRDD`, and `PolygonRDD`), implemented in this layer, allow the storage of different spatial objects. Furthermore, GeoSpark provides a Geometrical Operations Library which natively supports geometrical operations such as `Overlap()`, `MinimumBoundingRectangle()` and `Union()`.

– *SRDD Partitioning*. GeoSpark automatically partitions every SRDD using a global grid file. The algorithm for partitioning the SRDDs is as follows. First, a global grid file

is created by splitting the spatial space into a number of equal geographical size grid cells. Then, each element in the SRDD is assigned to every overlapping grid cell, i.e., if an element intersects with two or more grid cells, it is duplicated and different grid IDs are assigned to its copies.

– *SRDD Indexing*. *Spatial IndexRDD*s which inherit from SRDDs are implemented in GeoSpark to provide spatial indexes such as Quad-Tree [39] and R-Tree [48]. Furthermore, a local spatial index may be adaptively created on a SRDD partition to find an optimal trade-off between the run time performance and the memory/cpu usage in the cluster.

Spatial Query Processing Layer. Once geometrical objects are pre-processed and stored in the *Spatial RDD Layer*, users may invoke spatial queries (e.g., *Range Query*, *Join Query*) supported by this layer over large-scale spatial datasets. Query execution is parallelized in GeoSpark by using features such as partitioned SRDDs, spatial indexing, and fast *in-memory* computation. GeoSpark's algorithms for spatial range, spatial join, and *k*NN queries are described as follows.

– *Spatial Range Query*. The range query algorithm is executed by GeoSpark following the steps below.

1: Load target dataset
2: Partition data
3: Create a spatial index on each SRDD partition (optional)
4: Broadcast the query window to each SRDD partition
5: Check the spatial predicate in each partition
6: Remove duplicate spatial objects generated in data partitioning phase

– *Spatial Join Query*. The algorithm for processing spatial join queries in GeoSpark is given below.

1: Load two input SRDDs
2: Partition data
3: Create a spatial index on each SRDD partition (optional)
4: Join the two SRDDs by their keys (grid IDs)
5: Calculate spatial relations of spatial objects that have the same grid ID
6: Keep in the final results only the elements satisfying the spatial relation
7: Group results by grid ID
8: Remove duplicate results

– *Spatial kNN Query*. GeoSpark implements the following heap based top-k algorithm [87] to process spatial *k*NN queries.

1: Load a partitioned SRDD (*pSRDD*), a point (*P*), and a number (*k*)
2: **for** *partition* in *pSRDD* **do**
3: Calculate distances from the given point *P* to every object within *partition*
4: Maintain a local heap containing the nearest *k* objects around the point *P* based on the calculated distances
5: **end for**
6: Merge results from each partition

2.4.5 GeoTrellis

Geotrellis [43] is a high performance geoprocessing engine and programming toolkit. The main objective of GeoTrellis is the incorporation of geospatial analysis functionalities to real time interactive web applications. Focused on raster data processing, the following core problems are behind the development of GeoTrellis.

– Create scalable high performance geoprecessing web services.

– Parallelize geoprocessing operations to harness multi-core architectures.

– Create large-scale distributed geoprocessing services.

GeoTrellis helps developers to create simple, standard REST [38] services that return geoprocessing models results. These geoprocessing models are automatically parallelized and optimized. Both creating new operators and composing new operators with existing ones are easy tasks in GeoTrellis.

2.4.6 Magellan

Magellan [73] is a distributed execution engine for geospatial analytics on big data implemented on top of Spark. Modern database techniques are exploited to optimize geospatial queries. Once the application developer has written standard SQL or dataframe queries to evaluate geometric expressions, the execution engine efficiently lays data out in memory, picks the right query plan, and optimizes the query execution with efficient spatial indexes. Magellan supports multiple spatial data types (e.g., `Point`, `LineString`, `Polygon`, `MultiPoint`, `MultiPolygon`) and several spatial predicates (e.g., `Intersects`, `Contains`, `Within`). Spatial indexes in Magellan support the so called Z-Order curves [49] and are mainly used to speed up the spatial join performance.

2.4.7 LocationSpark

LocationSpark [95] is a spatial data processing system built as a library on top of Spark, providing spatial query APIs on top of the standard dataflow operators. The main features of LocationSpark are shown next.

- Support for spatial querying, spatial data updates, and spatial analytics. A rich set of spatial query operators (e.g., *spatial range*, *spatial kNN*, *spatial join*, and *kNN join*) is provided. LocationSpark supports data updates and spatio-textual operations. Moreover, spatial data analysis functions such as spatial data *clustering*, spatial data *skyline computation*, and spatio-textual *topic summarization* are provided by LocationSpark.
- Support for global and local *in-memory* spatial data indexes. Furthermore, an efficient spatial Bloom filter has been embedded into LocationSpark's indexes to avoid unnecessary network communication overhead.
- Tracking of frequently accessed spatial data and dynamic flushing of less frequently accessed data into disk.
- Storing spatial data as *key-value* pairs, where the key is a spatial geometric key (e.g., latitude-longitude value, line segment, polyline, rectangle, polygon) and the value type can be specified by the user (e.g., text type).

The layered system architecture of LocationSpark is depicted in Fig. 2.5. A detailed discussion of main layers of such architecture is provided below.

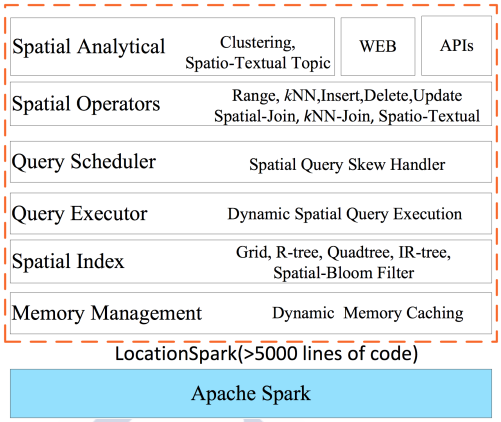


Figure 2.5: Architecture of LocationSpark.
source: *Tang et ál., in Proc. VLDB Endow., 2016* [95].

Query Scheduler. This layer is responsible for managing *query skew*[7] to mitigate runtime performance degradation of spatial queries. First, LocationSpark dynamically collects statistical information from each partition and detects hotspot data partitions. Then, in order to choose a set of partitions to be further reallocated to optimal workers, a cost model evaluates the overhead of repartitioning the hotspot partitions.

Query Executor. Specific query evaluation plans are executed in slave nodes once spatial queries and related data have been scheduled. For various alternative execution plans, LocationSpark evaluates the runtime and memory usage trade-offs. The best execution plan is selected and executed on each slave node.

Spatial Indexing. Two layers of spatial indexes (global and local) are provided by LocationSpark. The global index is responsible for partitioning data among worker nodes. Based on the underlying data distribution in space, the global index is built to ensure that each data partition has the same amount of data. A grid index and a region Quad-tree are provided as global indexes in LocationSpark. Furthermore, to match the needs of different scenarios, users can specify the type of the local index (e.g., grid local index, R-tree, a variant of the Quad-tree, or an IR-tree) to be executed on each data partition.

[7] Similarly to data skew, query skew occurs in a distributed computing environment when some queries are unevenly distributed in space and a number of data partitions are overwhelmed.

Memory Management. It is very common for spatial data analysis systems that certain partitions are queried more frequently than others. To deal with this issue, LocationSpark records access frequencies and corresponding time stamps in the spatial index. Then, access frequencies are aggregated to detect the most frequently accessed data. Finally, the most frequently accessed data is cached into memory and the less frequently used data is stored into disk.

2.4.8 Simba

Simba [106] is a scalable distributed *in-memory* analytics engine supporting efficient spatial queries and analytics over big spatial data. The main objectives of Simba are pointed out below:

- Simple and expressive programming interfaces.
- Low query latency.
- High analytics throughput.
- Excellent scalability.

Next, key features of Simba are highlighted:

- Support for rich spatial queries and analytics by extending Spark SQL [14] with core spatial operations. An expressive programming interface for these operations is offered in both SQL and DataFrame API.
- Support for spatial indexing to provide low query latency.
- Execution of multiple spatial queries in parallel to improve analytical throughput.
- Selection of good spatial query plans by using cost-based optimizations (CBO).
- Supply of novel algorithms for efficient and scalable execution of spatial operators.

The architecture of Simba, depicted in Fig. 2.6, shows the novel components added to the Apache Spark stack. A brief explanation of these components is provided below.

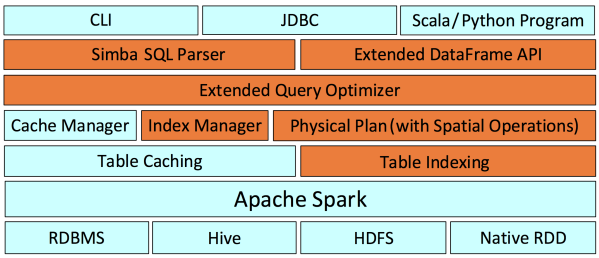


Figure 2.6: Architecture of Simba.
source: *Xie et ál., in Proc. SIGMOD, 2016* [106].

Simba SQL Parser and Extended DataFrame API. The Spark's DataFrame API and the Spark SQL's query parser have been extended with a set of spatial operators, and spatial keywords and grammar, respectively. Hence, users are allowed to express spatial queries using SQL-like statements or calling the novel spatial DataFrame API operators. Furthermore, Simba introduced index management commands in its programming interface similarly to that in traditional RDBMS.

Indexing. As previous works, Simba implements a two-layered indexing strategy to achieve better query performance. Several classic index structures such as hash maps, tree maps, and R-trees are implemented over Spark RDDs. Irrelevant partition pruning is possible through the collection of statistics from each RDD partition carried out by the global index. Local query processing inside each RDD partition is accelerated by local indexes in order to avoid scanning the entire partition. Index management commands allow users to build and drop indexes anytime on any table. The standard RDD structure has been extended with a new abstraction, called `IndexRDD`, that allows to persist indexes and associated data into disk and load them back to memory easily.

Spatial Operations. A number of spatial operations are supported only for point and rectangular objects. Simba provides different access and evaluation paths for each operation. Hence, both final users and Simba's query optimizer may choose the most appropriate method.

Extended Query Optimizer. A CBO module is implemented in Simba by extending the Catalyst optimizer of Spark SQL. This module leverages index support in Simba to make the best use of existing indexes and optimize complex spatial queries.

2.5 Distributed Spatio-Temporal Data Processing Systems

Big spatio-temporal data processing has become an important topic due to the huge amount of devices and applications providing large scale spatio-temporal data. Similarly to previous solutions for spatial data processing, current efforts in big spatio-temporal processing are built on top of the two aforementioned leading distributed large scale data processing frameworks, namely Hadoop [9] and Spark [109, 10]. Examples of implementations over MapReduce include a novel spatio-temporal indexing approach provided for the efficient processing of big array climate data [66], a query processing engine for massive trajectory data [70], a novel storage system for big spatio-temporal data analytics [94], and a system for querying and visualizing spatio-temporal satellite data [35]. ST-Hadoop [6] and Stark [92], two leading approaches based on Hadoop and Spark respectively, are discussed in the following subsections.

2.5.1 ST-Hadoop

ST-Hadoop [6] is a comprehensive extension to Hadoop and SpatialHadoop that provides native support for spatio-temporal data by injecting spatio-temporal data awareness inside each of their layers. ST-Hadoop replicates spatio-temporal index structures when loading data into HDFS. Consequently, aswers are retrieved by spatio-temporal queries with minimal data access. As we can see in Fig. 2.7, ST-Hadoop's architecture is largely similar to the SpatialHadoop's architecture. A description of its component layers is provided below.

- *Language layer*. The Pigeon [36] language has been extended to provide support for spatio-temporal data types (e.g., `Time`, `Interval`) and operations (e.g., `Overlap`, `Join`) over these new data types.
- *Operations Layer*. Two common spatio-temporal operations (*spatio-temporal range* query and *spatio-temporal join* query) are implemented in this layer.
- *MapReduce Layer*. The SpatialHadoop's MapReduce layer has been changed to allow ST-Hadoop to exploit its spatio-temporal indexes and to implement spatio-temporal predicates.
- *Indexing Layer*. Input files are indexed in ST-Hadoop following the next steps.
    1. *Sampling*. The spatial distribution of objects and how that distribution evolves over time is estimated by reading a random sample of the input data.

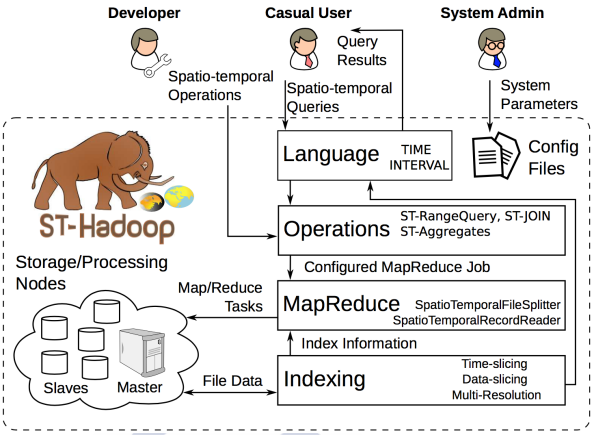


Figure 2.7: Architecture of ST-Hadoop.
source: *Alarabi et ál., in Proc. VLDB Endow., 2017* [6].

2. *Bulk loading*. The *spatio-temporal boundaries*[8] for the ST-Hadoop's index are built as follows. First, temporal boundaries are determined by dividing the *in-memory* sample into multiple intervals. Next, spatial boundaries are determined for data within each temporal slice. Finally, the two-level indexing is bulk loaded into a temporal hierarchy index. By default, the temporal hierarchy index provides four layers with a resolution of days, weeks, months, and years.
3. *Scanning*. ST-Hadoop scans the input file, physically partitions HDFS blocks, and assigns records to all overlapping partitions. The bulk loaded spatio-temporal boundaries, provided by step 2, determine the ST-Hadoop partitions.

2.5.2 Stark

Stark [92] is a framework for scalable spatio-temporal data analytics on Spark. By implementing different data types and several spatio-temporal operators (e.g., spatio-temporal *filter* and *join*) with various predicates, Stark is able to provide data analytics operations such as *kNN search* and *density based clustering*. The major features of Stark are identified below.

– A domain specific language (DSL) is provided to be seamlessly integrated into any Scala-based Spark program.

[8] Boundaries of a partition provided by a spatio-temporal partitioner represent the spatial region and time interval containing all elements within that partition.

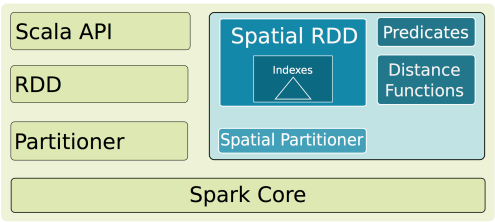


Figure 2.8: Architecture of Stark.
source: *Hagedorn et ál., in EDBT, 2017* [92].

– A set of spatio-temporal operators for *filter* and *join* with different predicates is provided.

– A *density based clustering* operator to find groups of similar events and a *kNN search* operator are implemented.

– Spatial partitioning and indexing make the execution of data analytics operators much more efficient and faster.

An overview of the Stark's architecture and its integration into Spark is shown in Fig. 2.8. Main components of that architecture are discussed next.

Partitioning. Currently, Stark only considers spatial dimensions for partitioning. Two different partitioning strategies are provided.

– *Grid Partitioner*. Data is divided into a number of intervals per dimension. Since Stark only takes into account 2D spatial dimensions for partitioning, this partitioner provides a grid of rectangular partitions with equal dimensions. In a first step, the bounds of these partitions are computed. Then, in a single pass over data, Stark uses these bounds to assign each element to the corresponding partition, i.e., the grid cell that contains this element.

– *Cost-based Binary Space Partitioner*. The Grid Partitioner do not takes into account the number of elements within each partition to split data among working nodes. Hence, that partitioning might result in unbalanced data distribution where some partitions contain the largest portion of data items while other partitions are empty. To solve this problem, this partitioner implements a cost-based binary space partitioning algorithm

based on [51]. First, data space is divided into two partitions with equal cost, where cost means number of elements. If the cost of one partition exceeds a threshold, it is recursively divided again in two partitions with equal cost. The recursion stops when the cost of a partition does not exceed the threshold or the algorithm reached the minimum side length for a partition. As a result, highly dense regions are split into multiple partitions while regions with low density belongs to the same partition.

When a spatial element spans across multiple partitions, it is assigned to only one partition (based on their centroid point) and the bounds of this partition are adjusted accordingly, resulting in overlapping partitions.

Indexing. Since Stark uses JTS [60] for spatial operations and JTS provides an R-tree implementation, Stark can use this index structure to index data within a partition. The user can choose between three different indexing modes.

- *No indexing*. The queried predicate evaluates every element within a partition.
- *Live Indexing*. The elements of each partition are put into an R-tree when the partition is being processed. As a result, a set of candidates whose minimum bounding box matches the required query is provided. The temporal predicate is evaluated, if needed, during this pruning process. Then, these candidates have to be checked for matching the query object.
- *Persistent Indexing*. When the same index may be used in subsequent executions, Stark allows to persist the index into disk/HDFS. This may avoid extra consuming time when creating the index.

DSL. Stark provides an DSL with several spatio-temporal operators that can be used for flexibly working with spatio-temporal data within any Scala-based Spark program. The `STObject` class enables the representation of spatio-temporal data in Stark. `STObject` has a field (`geo`) which stores the spatial attribute and an optional field (`time`) which stores the temporal information. When an RDD of 2-tuples (k, v) is defined and k is of type `STObject`, STARK implicitly creates the object `SpatialRDDFunction` that implements the spatio-temporal functions. Currently, Stark implements three spatio-temporal predicates (*intersects*, *contains* and *containedBy*), two spatial operators (*withinDistance* and *kNN search*), and a data

mining operator (*clustering*). The *clustering* operator implements the DBSCAN algorithm for Spark, based on MR-DBSCAN described in [51].

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

CHAPTER 3

GEODADIS

3.1 Introduction

According to the International Energy Agency (IEA), energy efficiency is "a mainstream tool for economic and social development", with potential "to support economic growth, enhance social development, advance environmental sustainability, ensure energy-system security and help build prosperity" [4]. To reach a significant improvement of energy efficiency in fishing vessels, the *Green Fish* project[1] [11] attempted to characterize the generation and consumption of energy during fishing activities. Different data acquisition systems [102] were developed and deployed in several fishing vessels.

To leverage the background on designing and deploying the previous data acquisition systems, GeoDADIS [100] enables data acquisition and data dissemination in in-situ sensor platforms. A wide range of technologies must be supported for data dissemination tasks. Data acquisition must fulfill the following requirements.

- Any sensor type must be supported.
- Any communication channel type must be supported.
- Addition of new sensors and new communication channels with a minimum effort.

The effort required to add new sensors that measure new parameters through new data acquisition channels must be minimum. Both synchronous and asynchronous data acquisition

[1] Founded by the Spanish and Galician public administrations.

channels must be supported. For the former, measures are pulled from the sensors by the framework. For the latter, sensors push measurements to the framework. Two special data acquisition channels used to (1) provide the geographic location of the platform and (2) synchronize the clock, must be specified by the system configuration. Furthermore, additional metadata must be stored in system configuration to enable the specification of the range of historic recorded measurements and the frequency of the sampling process, for each observed parameter.

Both client/server (i.e., services query the framework to pull data) and publish/subscriber (i.e., the framework pushes data to the services) data services must be supported. Similarly to data acquisition channels, a minimum effort in the implementation of new data services is required. New remote administration services must be implemented with a minimum effort as well.

A minimum effort is also required in the implementation of new data storage technologies for distinct measured parameters. Notice for example the different storage capabilities required by a single temperature value and a complex satellite image.

The remainder of this Chapter is organized as follows. Section 3.2 provides a detailed description of the general layered architecture of GeoDADIS, introducing the relevant functionality of each layer and corresponding components. An in-depth description of main components of GeoDADIS architecture is given in Section 3.3. Thus, relevant details about structure and functionality of components DataDissemination, DataAcquisition, ConfigurationManager and DataManager are provided in Section 3.3.1, Section 3.3.2, Section 3.3.3, and Section 3.3.4, respectively. Finally, Section 3.4 introduces an experimental implementation of GeoDADIS developed to monitor people health status in educational environments.

3.2 System Architecture

The general component architecture of GeoDADIS, Fig. 3.1, is composed of three main software layers. General purpose functionality related to system control, sensor data management and configuration metadata management is provided by three main components in layer *Data and Control Management*. *DataManager* provides persistent storage functionality required by component *DataAcquisition* and data query functionality demanded by component *DataDissemination*. Component *ConfigurationManager* enables the remainder components to access

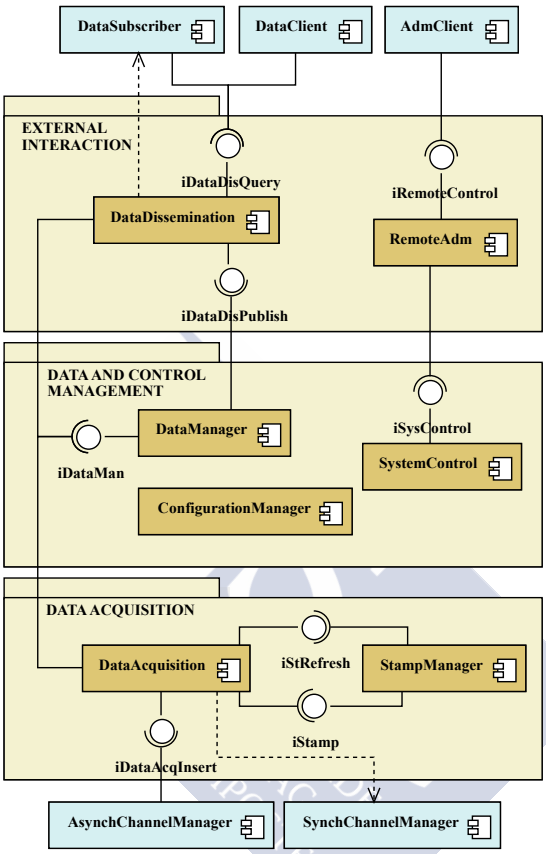


Figure 3.1: GeoDADIS component architecture.

configuration settings. Functionality enabling to start up, stop and restart different components is provided by *SystemControl*.

External data acquisition channels enable heterogeneous sensors to provide measures for the sampling processes implemented in layer *Data Acquisition* at the bottom of GeoDADIS architecture, Fig. 3.1. For each measured parameter of each sensor, administration staff configure both the time range and the frequency of the sampling process. Regarding trigger properties of available sensors, the proposed architecture enables both the time-triggered approach (i.e., sampling frequency is determined by the system) and the event-triggered approach (i.e., sensors deliver measures to be sampled independently of the system). As shown in Fig. 3.1,

each data acquisition channel must be associated to an external channel manager component. *AsynchChannelManagers* enable the communication with event-triggered sensors whereas *SynchChannelManagers* enable to query time-triggered sensor at configured sampling rate. Every new measure generated by an event-triggered sensor is delivered to GeoDADIS by the relevant *AsynchChannelManager* through the *iDataAcqInsert* interface. A buffer located in the *DataAcquisition* component temporarily records the last measure of each sensor. GeoDADIS queries SynchChannelManagers of time-triggered sensors to sample new measures at relevant sampling rate. A spatio-temporal stamp (i.e,. a time value together with the geographic location of the platform) is requested to *StampManager* through interface *iStamp* for each sampled measure. Then, interface iDataMan of component *DataManager* is used to deliver the sampled measure and relevant time stamp. Component *StampManager* periodically requests the current value of the spatio-temporal stamp by using the interface *iStRefresh*. Location and time of stamps are provided by sensors, thus external data acquisition channels must be used to obtain such values. Configuration data provided by *ConfigurationManager* must store the refresh rate and the data acquisition channels used to get the stamp components.

Users and administration staff interaction with GeoDADIS is enabled by the functionality provided by the layer *External Interaction*. Component *DataDissemination* is responsible for the proper communication between the *DataManager* and the available data services. Two different communication approaches are enabled here. The client/server approach enables communication with *DataClients* whereas publish/subscribe approach is used in communication with *DataSubscribers*. Notice that both *DataClient* and *DataSubscriber* data services are external to GeoDADIS. Interface *iDataDisQuery* is used by *DataClients* to deliver queries over stored measures. *DataDissemination* delegates such queries to *DataManager* through interface *iDataMan*. Component *DataManager* notifies the component *DataDissemination* through interface *iDataDisPublish* every time a new measure is recorded. Then, the new measure is delivered to appropriate *DataSubscribers* which in turn may use interface *iDataDisQuery* to request queries over stored measures. Configuration and control functionality is provided to administration staff by component *RemoteAdmin* which delegates actual requests to *ConfigurationManager* and *SystemControl* respectively.

3.3 Main Components

Prior to the detailed description of GeoDADIS' main components, a brief introduction to three well known design patterns [41] and how they relate to GeoDADIS is provided below.

- *Singleton pattern*: used to enable coordinated access to shared resources by restricting the instantiation of a class to only one object. Configuration data, data acquisition channels, data services and control clients are examples of such resources in GeoDADIS.
- *Adapter (wrapper) pattern*: translates one interface for one class into a compatible interface to enable the cooperation of classes implementing different interfaces. GeoDADIS uses adapters to uniformly access the specific interfaces of control clients, data services and data acquisition channels.
- *Observer (publish/subscriber) pattern*: notifies changes in the state of an object (subject) to a list of objects (observers). The publish/subscribe communication between component *DataDissemination* and component *DataSubscriber* implements this pattern in GeoDADIS.

3.3.1 *DataDissemination*

Component *DataDissemination* combines the three above design patterns to provide flexible external data access. Both data clients, following a client/server approach through the *iDataDisQuery* interface, and data subscribers, following a publish/subscribe protocol, may access the *DataDissemination* component. By using the *Adapter* design pattern, the effort to add a new data service is restricted to the implementation of a new data subscriber or data client adapter class. The *Singleton* design pattern is used to coordinate the access to the available data services.

The main class of the component (*DataServicesManager*) implements the three interfaces of *DataDissemination*, Fig. 3.2. Pseudocode describing the implementation of relevant methods of the interfaces that enable the querying (*iDataDisQuery*)and publishing (*iDataDisPublish*) of measures is depicted in the figure as well. Methods *getParameters* and *getMeasures* of the interface *iDataDisQuery* enable external data service components to query the recorded data. Implementation of such methods is delegated to relevant methods of components *ConfigurationManager* and *DataManager* respectively. The implementation of method *publishMeasure* of interface *iDataDisPublish*, which is used by component *DataManager* to deliver

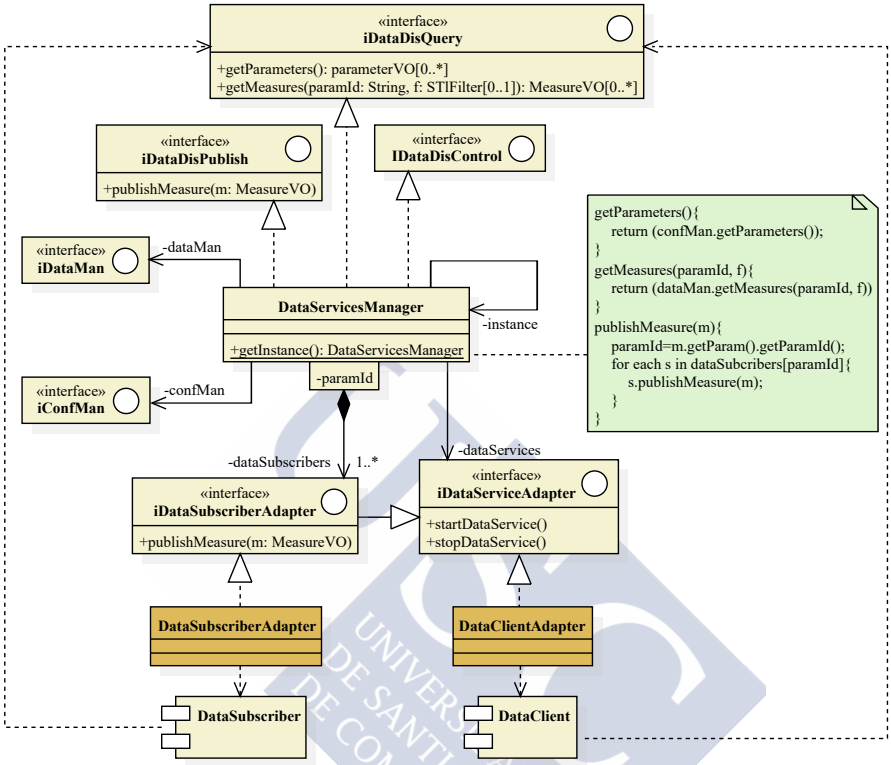


Figure 3.2: UML Class Diagram of component DataDissemination.

measures to appropriate external data subscribers, follows a combination of the *Observer* and *Adapter* design patterns. When a new measure is received, *DataServicesManager* (subject class of the *Observer* pattern) uses the measures' *paramId* to notify appropriate *DataSubscriberAdapters* (both *Observer* class and *Adapter* class) by calling method *publishMeasure* of interface *iDataSubscriberAdapter*. The list of *DataSubscriberAdapter* names for each *paramId* as well as the name of the *DataClientAdapter* for each data service is part of the component configuration metadata. *Adapter* classes are also required to enable GeoDADIS to access control functionality (start, restart and stop) of data clients.

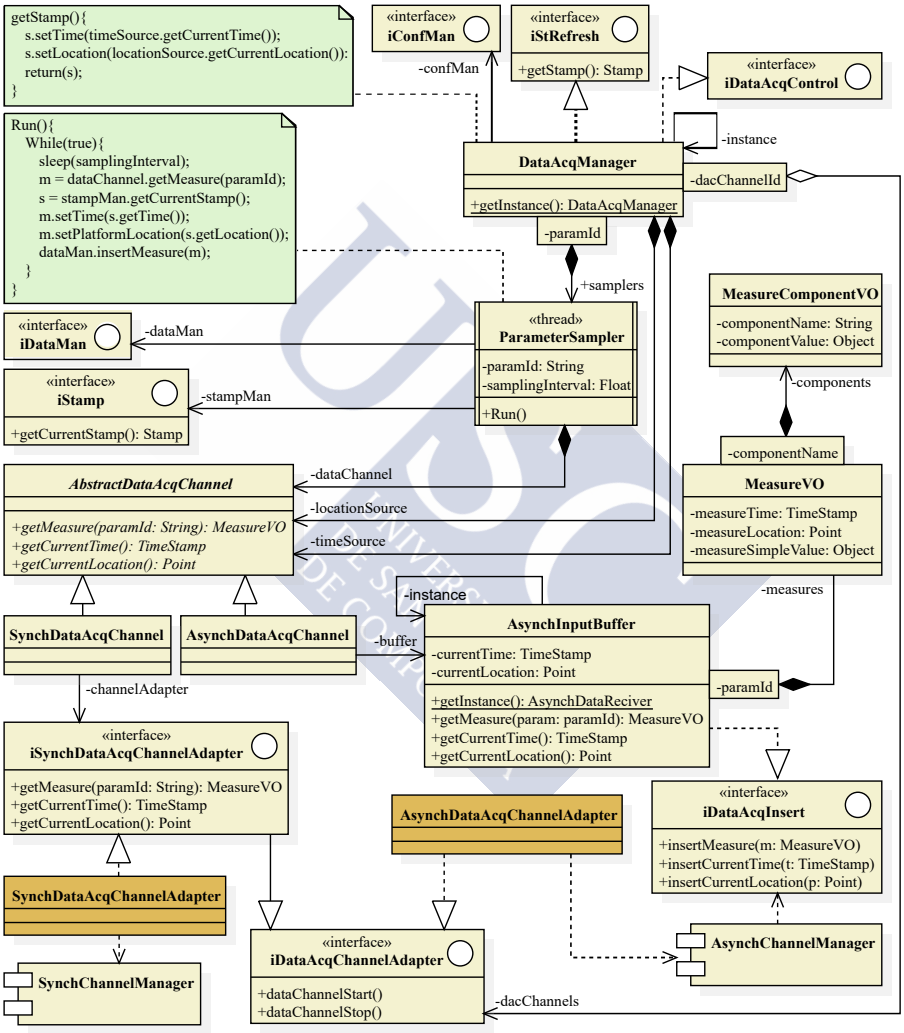


Figure 3.3: UML Class Diagram of component DataAcquisition.

3.3.2 *DataAcquisition*

Component *DataAcquisition* provides general purpose functionality to enable data acquisition over both synchronous and asynchronous data communication channels. Similarly to *DataDissemination* component, the effort to add a new channel is restricted to the implementation of a new data acquisition channel adapter class due to the use of the *Adapter* design pattern.

The functionality of the *DataAcquisition* component (see Fig. 3.3 for a graphical representation of its internal structure) is accessed through the *DataAcqManager* class. Interface *iDataAcqControl* provides component control functionality whereas interface *iStRefresh* is used to obtain the current spatio-temporal stamp from the appropriate data acquisition channels. The implementation of the class follows a *Singleton* design pattern in order to coordinate the concurrent access to both the sampling threads and the data acquisition channels. As it is shown in pseudocode given in the figure for class *DataAcqManager*, the implementation of method *getStamp* of the interface *iStRefresh* access directly the data channels (class *AbstractDataAcqChannel*) configured as location and time sources. Regarding the data acquisition process, each sensed parameter is sampled by a different *ParameterSampler* thread. This implementation is also illustrated with pseudocode in the figure. First, the thread sleeps during a given *samplingInterval* that is obtained from the configuration data of the specific parameter. After waking up, the thread uses its data acquisition channel, which is also obtained from the configuration data, to obtain the next measure of the parameter. Next, the current stamp obtained from the *iStamp* interface is used to associate current time and location to the measure. Finally, the stamped measure is delivered to the component *DataManager* through interface *iDataMan*.

A data acquisition channel (*AbstractDataAcqChannel*) may access measures of either a synchronous (*SynchDataAcqChannel*) or an asynchronous (*AsynchDataAcqChannel*) external channel manager component. Measures of synchronous channel managers are accessed directly through a relevant adapter class (*SynchDataAcqChannelAdapter*), which is obtained from configuration data and that implements the interface *iSynchDataAcqChannelAdapter*. On the other hand, measures of asynchronous channel managers are obtained from a buffer (*AsynchInputBuffer*), that is populated with the last measure of each parameter through the interface *iDataAcqInsert*. Coordinated access to the buffer is achieved through the use of the *Singleton* design pattern for its implementation. Notice that an adapter class is still required

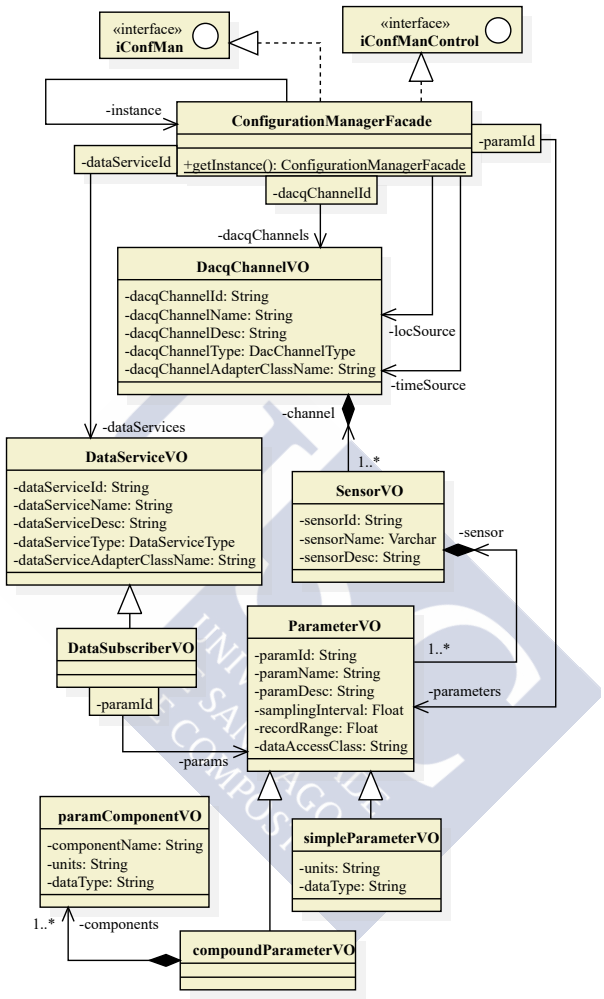


Figure 3.4: UML Class Diagram of component ConfigurationManager.

for asynchronous channel managers in order to access their control functionality (start, stop and restart) from GeoDADIS.

3.3.3 *ConfigurationManager*

The internal structure of the *ConfigurationManager* is shown in the UML class diagram of Fig. 3.4. The *Singleton* design pattern enables the *ConfigurationManagerFacade* to provide coordinated access to the available configuration data through interface *iConfMan*. Starting, stoping and restarting the component may be done through interface *iConfManControl*. Configuration data consists of the following three collections.

- *parameters (ParameterVO class)*: each measured parameter has *samplingInterval* and *recordInterval* properties. The former is used by component *DataAcquisition* to determine its sampling frequency. The latter is used by component *DataManager* to determine when a recorded measure is old enough to be deleted. The sensor and, therefore, the data acquisition channel from which the parameter has to be sampled are also recorded, as it is shown in the figure. Property *dataAccessClass* records the name of the data access object class that component *DataManager* has to use to access the parameters data persistence functionality. Finally, it is noticed that both simple parameters (*simpleParameterVO* class) like temperature and compound parameters (*compoundParameterVO* class) like wind (it has speed and direction components) are supported.

- *dataChannels (DacqChannelVO class)*: each data acquisition channel has a property *dacqChannelType* that determines whether the channel is synchronous or asynchronous. Besides, property *dacqChannelAdapterClassName* records the name of the adapter class that component *DataAcquisition* has to use to interact with the external channel manager. The data acquisition channels that are used as location and time sources are determined by associations with respective roles *locSource* and *timeSource*.

- *dataServices (DataServiceVO class)*: each data service has a *dataServiceType* (either data client or data subscriber). The list of parameters about which each data subscriber (*DataSubscriberVO*) has to be notified is also recorded. Property *dataServiceAdapterClassName* records the name of the adapter class that component *DataDissemination* has to use to interact with the external data service component.

3.3.4 *DataManager*

The functionality of component *DataManager*, Fig. 3.5, is accessed through the class *DataManagerFacade*. More precisely, interface *iDataManControl* provides component control

functionality and interface *iDataMan* enables inserting and retrieving measurements from persistent storage. In particular, operation *getMeasures* is used to obtain the recorded measures of the parameter identified by *paramId* for which the spatio-temporal filter f holds. Such a filter enables the combined use of both the interval predicates defined by [7] and the spatial predicates defined by [27]. Retrieved measures include both measured value (various values in case of compound parameters) and spatial and temporal stamps. On the other hand, operation *insertMeasure* records a measure in persistent storage and delivers it to component *DataDissemination* through interface *iDataDisPublish* (in order to notify relevant subscribers). Data insertions and queries of measures of a given parameter are executed by its relevant class *ParameterDAO*, enabling this way different implementations for different parameters. The pseudocode given in the figure for the operation *insertMeasure* of the *DataManagerFacade* illustrates the use of the collection of *ParameterDAO* classes and the interface *iDataDisPublish*.

3.4 Experimental Implementation

A solution for heterogeneous sensor data integration in crowdsensing applications applied to health monitoring in educational environments using low cost hardware has been proposed in [101]. To track common respiratory diseases among members of the educational community, an application has been developed to collect data from occupants of several educational buildings. Different collection methods, protocols and standards have been used. Measurements of environmental parameters such as temperature, humidity and occupancy have been obtained from buildings through the Building Management System using the BACnet protocol. External environmental measurements have been obtained in the building surrounding areas through meteorological community services using the Observations and Measurements data exchange protocol. The HL7 protocol has been used with a low cost set of medical sensors to collect heath data, e.g., blood oxygen, pulse, body temperature, breathing status.

Based on the GeoDADIS architecture, the data platform architecture depicted in Fig. 3.6 has been implemented in [101] to enable the integrated collection of above data. Similarly to GeoDADIS, the *Adapter* design pattern has been adopted to access specific interfaces of data acquisition channels and data services in a uniform manner, and the Observer design pattern has been used to enable publish/subscribe communication with the Analysis Application developed in [101]. Unlike GeoDADIS, the implemented platform does not provide

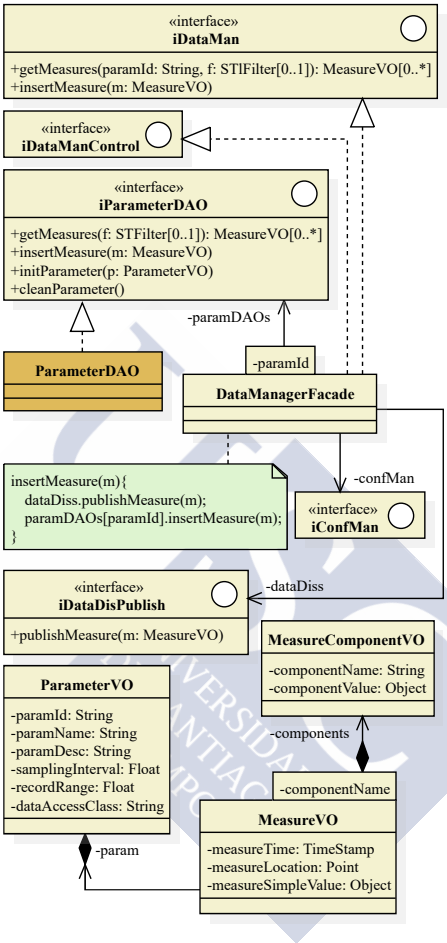


Figure 3.5: UML Class Diagram of component DataManager.

in-situ data acquisition, control administration nor spatio-temporal stamping of sampled observations. Provision of data acquisition devices is a feature of this platform that is not present in GeoDADIS.

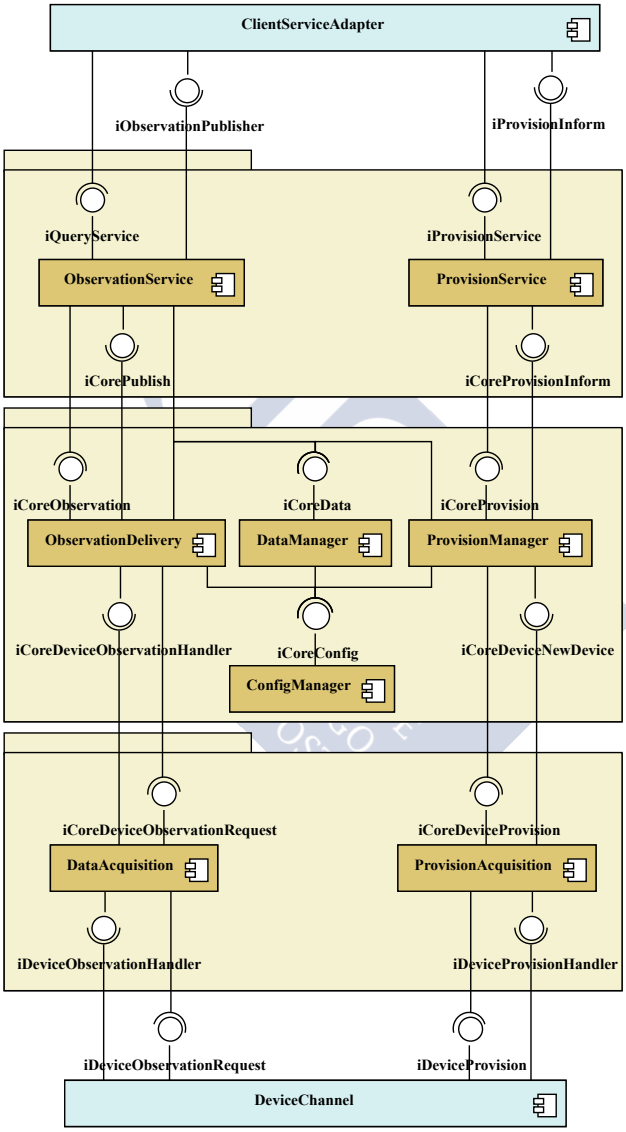


Figure 3.6: Component architecture of experimental implementation.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

CHAPTER 4

SODA DESIGN

4.1 Introduction

The design of the Spatio-Temporal Observation Data Management System (SODA) [103] mentioned in Section 1.4 is described in detail in this chapter. The problem description provided in Section 1.2 and the objectives stated in Section 1.4 serve as a basis for the definition of the requirements that SODA must fulfill. Thus, it is easy to see that an Observation Data Management System such as SODA should be composed of two major components.

An *observation data warehouse* enables the storage of observation data, provides the required observation semantics and facilitates the representation of *Entity* and *Sampled* data in an integrated manner, i.e., using the same data structures. A spatio-temporal data model has been defined to provide the novel data types and data structures required for an efficient integrated representation and storage of temporal, spatial and spatio-temporal components of observation data from a functional point of view. On top of this underlying data model, an observation data model has been defined to provide observation semantics. A novel XML-based language, called XODDL, has been defined to enable the definition of the observation data warehouse schema.

An *observation data analysis* system enables the analysis of observation data through the proposed set of operations. A novel hybrid logical-functional language, called MAPAL, is proposed to enable the analysis and processing of observation data. Specific syntax is also proposed to define *internal* processes, i.e., those processes executed within SODA during ETL tasks to generate new observation data from imported data or previously generated and

stored data. Finally, required operators must be defined to actually perform the analysis tasks defined by users though MAPAL queries.

The remainder of this chapter is organized as follows. Section 4.2 is devoted to the definition of the Observation Data Warehouse in SODA. Spatio-temporal and observation data models are described in Section 4.2.1 and Section 4.2.2, respectively. The observation data analysis system proposed in SODA is explained in Section 4.3. First, MAPAL language is fully described in Section 4.3.1. Then, the proposed syntax to define *internal* processes is explained in Section 4.3.2. Next, Section 4.3.3 defines the required operators to perform observation data analysis. And finally, Section 4.3.4 shows an example of how a MAPAL query is translated into a sequence of SODA operators.

4.2 Observation Data Warehouse

Based on general functionalities for an observation data management system specified in Section 1.2, an observation data warehouse should meet the following requirements.

- Support for the representation of data coming from the integration of temporal, spatial and spatio-temporal *samplings* with classical E/R data.
- Direct support for observation semantics, through the representation of appropriate required metadata (introduced in Section 1.1), must be provided.
- System data types must enable the representation of temporal and spatial data with parametric resolution. Additionally, implicit and explicit castings should be provided to ease the transformation between these resolutions.

From the above requirements, an underlying spatio-temporal data model is first defined. Capabilities of this data model go beyond those of an observation data model and enable the processing of any type of spatio-temporal data. To build the definitive data model for the observation data warehouse, structures for observation metadata are then added on top of such underlying data model.

4.2.1 Spatio-temporal Data Model

It is well known that relational formalism applied to sampled data processing results in highly inefficient approaches. On the other hand, functional models, which fit well sampled data,

have already been used to manage E/R data in the area of Functional Databases [45]. A novel spatio-temporal data model based on the well known mathematical concept of *function* is defined in this section. Definition and storage of various data types (conventional, temporal and spatial), functions to manipulate data values (*Intensional Mappings*), functions to record data values (*Extensional Mappings* and *Extensional MappingSets*), and data singletons (*Constants*) are supported.

Data types

Conventional data types

They consist of the data types usually supported by general purpose data management systems, including *Boolean*, *CString* (variable size character string), *Integer*, and *Real*. In scientific applications, the user knowledge about the precision and scale of numeric data is very important to choose the most appropriate physical representation for real numbers. Hence, a fixed point numeric representation is supported by the parametric data type *FixedPrecision(P,S)*, with conventional semantics for P (precision, i.e., maximum of number of decimal digits) and S (scale, i.e., number of decimal digits in the fractional part). Default and maximum values for P and S are system defined: *DP* (default P), *DS* (default S), *MP* (maximum P), and *MS* (maximum S). Every value N of type *FixedPrecision(P, S)* may be written in the form

$$N = n \cdot 10^{-S}$$

where n is the integer value in the range $(-10^P, 10^P)$ that actually will be stored, together with P and S. Thus, P determines the underlying primitive[1] integer data type used to store n.

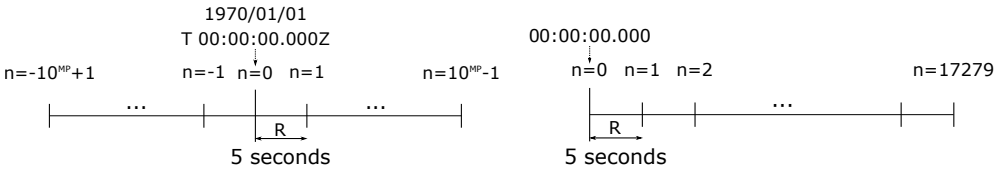
Additionally, all data types enable the representation of a special *undefined* value denoted by $\perp$.

Temporal data types

TimeInstant(R)

$$\left\{ t = n \cdot R \mid n \in \mathbb{Z};\ -10^{MP} < n < 10^{MP} \right\} \cup \{\perp\}$$

[1] Considered primitive data types are: *byte*, *short*, *int* and *long*.



(a) TimeInstant(5). (b) Time(5).

Figure 4.1: Example of *TimeInstant(R)* and *Time(R)* data types where $R = 5s$.

Time(R)

$$\left\{ t = n \cdot R \mid n \in \mathbb{Z};\ 0 \leq n \cdot R < 24\ hours \cdot 3600\ \frac{seconds}{hour} \right\} \cup \{\perp\}$$

Date(R)

$$TimeInstant(86400)$$

The above temporal data types have been defined to enable the representation of discrete multi-resolution time values, where R (temporal resolution) is a value of data type *Double* and n is the corresponding index in the defined temporal sampling. Similarly to *FixedPrecision* values, only R and n values are stored[2]. Notice that the discrete temporal value $t_1 = n_1 \cdot R$ actually represents the continuous time interval defined by the following set of instants

$$\{\, t \mid n_1 \cdot R \leq t < (n_1 + 1) \cdot R \,\}$$

The semantics of a *TimeInstant* value is a positive or negative time shift in seconds from an absolute reference time instant, $t = 0$. The most commonly used value for this reference time instant in current DBMS, and also in SODA, is `1970-01-01 T 00:00:00.000000Z`.

The semantics of a *Time* value is a positive time shift in seconds from a relative reference time instant, $t = 0$. The value used for this time instant in SODA is the beginning of each day in civil time throughout the world, i.e., `00:00:00.000000`.

As an example, the available values that can be represented by data types *TimeInstant*(5) and *Time*(5) are depicted in Fig. 4.1(a) and Fig. 4.1(b), respectively.

[2] The primitive integer data type used to store temporal values is *long*.

Spatial data types

Point1D(P,R)

$$\left\{ x = n_x \cdot R \mid n_x \in \mathbb{Z};\ -10^P < n_x < 10^P \right\} \cup \{\perp\}$$

Point2D(P,R)

$$\left\{ (x,y) = (n_x \cdot R,\ n_y \cdot R) \mid n_x, n_y \in \mathbb{Z};\ -10^P < n_x, n_y < 10^P \right\} \cup \{\perp\}$$

To enable users to represent discrete multi-resolution spatial values, the above spatial data types have been defined, where P (precision) is of type *Integer* and R (spatial resolution) is of type *Double*. Notice that the discrete *Point1D* value $x_1 = n_1 \cdot R$ actually represents the continuous 1D spatial interval

$$\left\{ x \mid x \in \mathbb{R};\ x_1 - \frac{R}{2} \leq x < x_1 + \frac{R}{2} \right\}$$

and the discrete *Point2D* value $(x_1, y_1) = (n_{x_1} \cdot R,\ n_{y_1} \cdot R)$ represents the following 2D rectangle

$$\left\{ (x,y) \mid (x,y) \in \mathbb{R}^2;\ x_1 - \frac{R}{2} \leq x < x_1 + \frac{R}{2};\ y_1 - \frac{R}{2} \leq y < y_1 + \frac{R}{2} \right\}$$

Similarly to *time instant* values, a *Point1D* value is a 1D positive or negative spatial shift in meters from a specific *origin* point, $x = 0$. In fact, *Point1D(P,R)* data type defines a 1D spatial sampling in $\mathbb{R}$ and provides a 1D cartesian coordinate system. Fig. 4.2 depicts feasible values, n_x, for *Point1D*(1,1) and *Point1D*(1,0.5).

Likewise, a *Point2D* value is a positive or negative 2D spatial shift in meters from a specific *origin* point, $x = (0,0)$. *Point2D(P,R)* data type defines a 2D spatial sampling in $\mathbb{R}^2$ and provides a 2D Cartesian coordinate system. Fig. 4.3 depicts all possible values, (n_x, n_y), for *Point2D*(1,1) and *Point2D*(1,0.5).

Similarly to previous data types, only P, R and integer indexes n_i are stored. As *FixedPrecision* values, P determines the underlying primitive integer data type used to store the integer index value.

Geometric data types

Based on *Point2D(P,R)* data type and on the standard specification defined in [58], the following data types enable the modeling of geometries in 2D euclidean spaces:

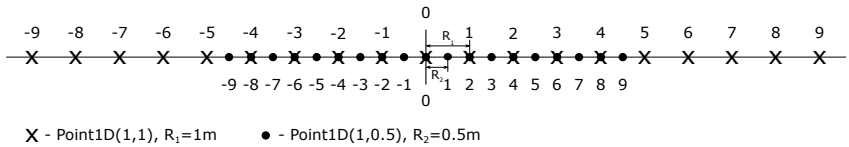


Figure 4.2: Spatial data types *Point1D*(1,1) and *Point1D*(1,0.5).

- *LineString(P,R)*: vector polylines defined by sequences of elements of *Point2D(P,R)*.
- *Polygon(P,R)*: vector polygons, possibly with holes, whose borders are defined by sequences of elements of *Point2D(P,R)*.
- *GeometryCollection(P,R)*: heterogeneous collections of geometries of any of the following data types: *Point2D(P,R)*, *Polyline(P,R)* and *Polygon(P,R)*.
- *MultiPoint(P,R)*: homogeneous collections of *Point2D(P,R)* geometries.
- *MultiLineString(P,R)*: homogeneous collections of *LineString(P,R)* geometries.
- *MultiPolygon(P,R)*: homogeneous collections of *Polygon(P,R)* geometries.
- *Geometry(P,R)*: abstract type that enables the representation of geometries or geometry collections of any of the above 2D data types.

Data Structures

The following data structures, *Dimensions*, *Extensional MappingSets* and *Constants* enable the modeling and recording of spatio-temporal *entity* and *sampled* data.

Dimensions

A *Dimension* is a finite set of elements of a given data type. More formally, a *Dimension* d over data type T, denoted $d : T$, is defined as a non-empty finite subset of $T - \{\perp\}$. *Dimensions* may be defined only over conventional, temporal and spatial data types, and not over geometric data types.

Temporal and spatial *Samplings* are special cases of *Dimensions* of major interest for the modeling of sampled spatio-temporal data. Thus, if *min* and *max* are two values of the same

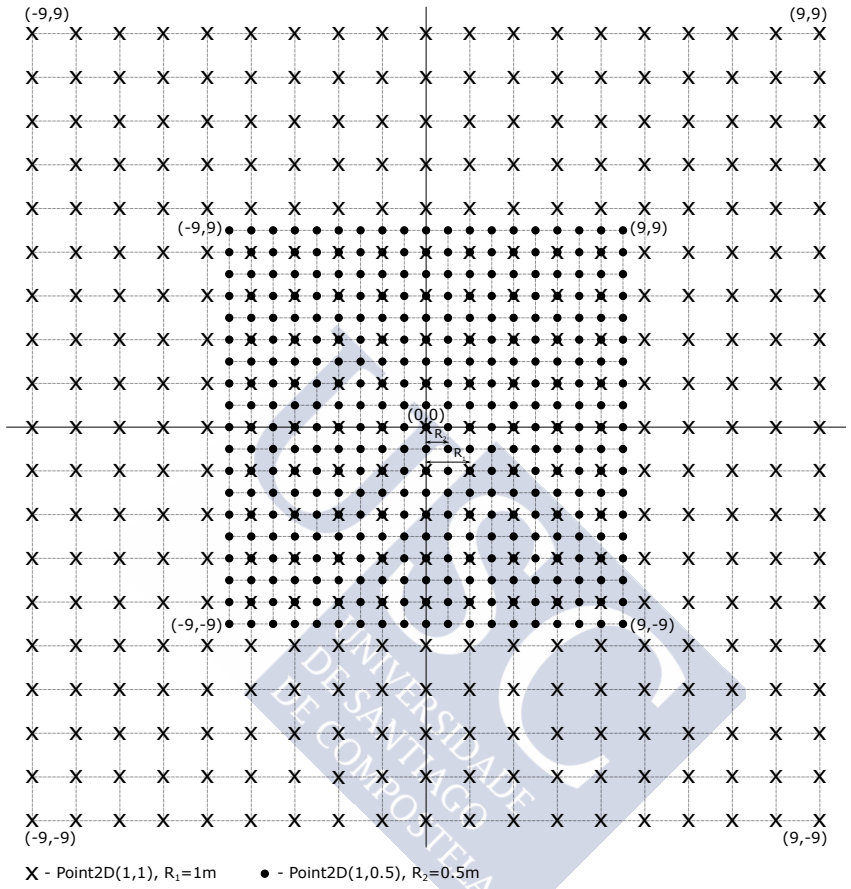


Figure 4.3: Spatial data types *Point2D*(1,1) and *Point2D*(1,0.5).

Time, *TimeInstant*, *Date* or *Point1D* data type T where $min < max$, then a *1D Sampling S* from *min* to *max*, denoted $S(min, max)$, is defined as the following *Dimension* over T

$$S(min, max) = \{\, s \in T \mid min \leq s \leq max \,\}$$

Likewise, if $s_m = (x_m, y_m)$ and $s_M = (x_M, y_M)$ are two values of the same *Point2D* data type T where $x_m < x_M$ and $y_m < y_M$, then a *2D Sampling S* from s_m to s_M, denoted $S(s_m, s_M)$, is defined as the following *Dimension* over T:

$$S(s_m, s_M) = \{\, (x,y) \in T \mid x_m \leq x \leq x_M,\ y_m \leq y \leq y_M \,\}$$

Notice that, in general, a *Dimension* is stored by the *explicit* recording of each of its elements. However, *Samplings* are *implicitly* recorded by the storage of their limit and parametric values.

Extensional MappingSets

An *Extensional MappingSet* is a finite set of mappings, *Extensional Mappings*, with a common domain defined by the Cartesian product of *Dimensions*.

If $d_1, d_2, \ldots, d_n$ is a sequence of not necessarily distinct *Dimensions* and T is a data type, then an *Extensional Mapping* with signature $M(d_1, d_2, \ldots, d_n) : T$ is defined as a function $M : d_1, d_2, \ldots, d_n \rightarrow T$.

An *Extensional MappingSet* with signature $EM(d_1, d_2, \ldots, d_n \mid M_1 : T_1, M_2 : T_2, \ldots, M_m : T_m)$ is defined as the following finite set of *Extensional Mappings*:

$$EM(d_1, d_2, \ldots, d_n \mid M_1 : T_1,\ M_2 : T_2,\ \ldots\ ,\ M_m : T_m) = \\ \{\ M_1(d_1, d_2, \ldots, d_n) : T_1,\ M_2(d_1, d_2, \ldots, d_n) : T_2,\ \ldots\ ,\ M_m(d_1, d_2, \ldots, d_n) : T_m\ \}$$

An *Extensional MappingSet EM* is extensionally defined by a finite set of nested tuples of the form (d, m), where $d \in d_1 \times d_2 \times \ldots \times d_n$ and $m \in T_1 \times T_2 \times \ldots \times T_m$.

Constants

A *Constant C* of type T, denoted by $C : T$, is defined as an atomic value of type T.

Following a functional database approach [45], *Dimensions* and *Extensional MappingSets* enable the modeling of *Entities* and *Relationships* between them. Hence for example, *Dimensions* `StationId` and `MunCode` in Fig. 4.4 record identifiers and codes of weather stations and municipalities, respectively. The remainder properties of stations and municipalities are modeled by relevant *Extensional MappingSets* `Station` and `Municipality`.

Beyond classical ER data, this model integrates the representation of temporal and spatial 1D and 2D sampled data. For example, *Dimensions* `ObsData` and `Loc5m` in Fig. 4.4 are respectively a temporal *Sampling* and a 2D spatial *Sampling*. These *Samplings* are used to model the *phenomenonTime* of temperature, humidity and wind speed observations at each weather station in Extensional MappingSet `Observation`, and the *geolocation* of elevation observations in Extensional MappingSet `Topo`.

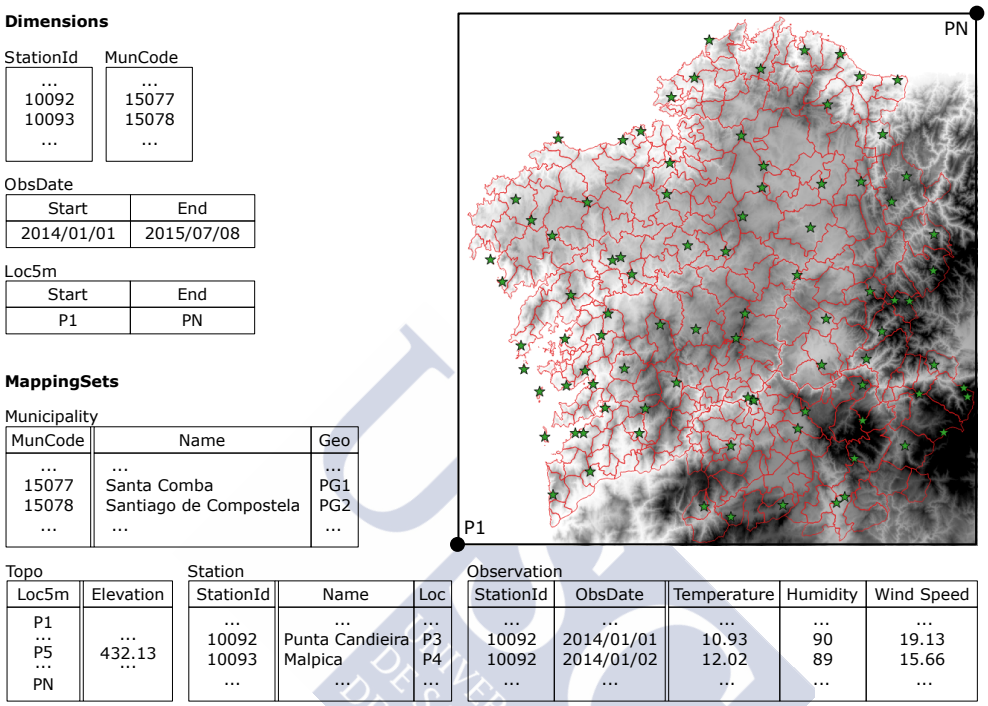


Figure 4.4: Data Structures.

Image in Fig. 4.4 depicts the following geolocated *Extensional Mappings*: `Station.Loc` (green starred locations), `Municipality.Geo` (red line geometries) and `Topo.Elevation` (gray-scale raster).

Intensional Mappings

An *Intensional Mapping* is a function defined over the available data types, either by an algorithm or an analytical expression.

If $T_1, T_2, \ldots, T_n$ is a possibly empty sequence of not necessarily distinct data types and T is also a data type, then an *Intensional Mapping* with signature $M(T_1, T_2, \ldots, T_n) : T$ is defined as a function $M : T_1, T_2, \ldots, T_n \rightarrow T$.

Primitive intensional mappings. Defined by algorithms, *primitive mappings* may be already incorporated into the system or provided by the user through user-defined functions.

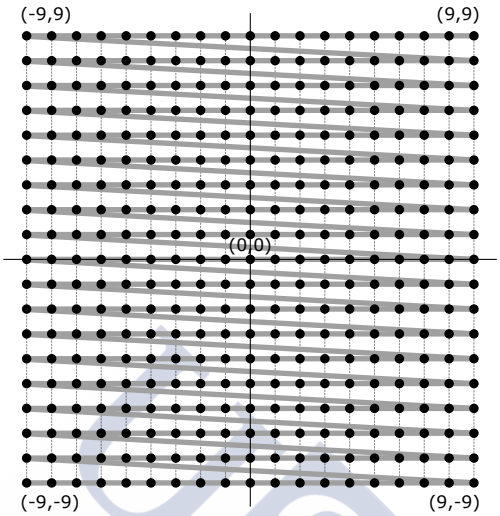


Figure 4.5: *Point2D* Space Filling Curve.

They include conventional, temporal and spatial functions like those supported by SQL and relevant extensions [58]. Comparison and arithmetic operators are also supported and defined even for temporal and spatial data types. Fig. 4.5 depicts the specific space filling curve used in SODA for defining a total ordering in *Point2D* data type. Implicit type castings are automatically applied between data types of the same family during the evaluation of functions and operations. For illustration purposes, primitive intensional mappings defined for all MAPAL data types are shown in Table 4.1. A complete list of intensional mappings defined for specific data types is provided in Appendix A.

Argument data types must be compatible for the underlying operation or function to successfully execute each mapping. Thus, overloaded mappings (i.e., different argument versions of each mapping) have been defined for each data type in SODA. For instance, the overloaded mappings defined in SODA for mapping $equal(o_1, o_2)$ are described below.

equal(*Boolean* b_1, *Boolean* b_2): returns the *Boolean* value `true` iff $(a \wedge b) \vee (\bar{a} \wedge \bar{b})$.

equal(*CString* s_1, *CString* s_2): returns the *Boolean* value `true` iff s_1 is lexicographically equal to s_2, i.e., represents the same sequence of char values.

Primitive mapping	Description
$distinct(o_1, o_2)$	Returns `true` if $o_1 \neq o_2$ and returns `false` otherwise.
$equal(o_1, o_2)$	Returns `true` if $o_1 = o_2$ and returns `false` otherwise.
$getDataType(o)$	Returns the data type of o.
$greaterThan(o_1, o_2)$	Returns `true` if $o_1 > o_2$ and returns `false` otherwise.
$greaterThanOrEqualTo(o_1, o_2)$	Returns `true` if $o_1 \geq o_2$ and returns `false` otherwise.
$isDefined(o)$	Returns `true` if o is not `null` and returns `false` otherwise.
$lowerThan(o_1, o_2)$	Returns `true` if $o_1 < o_2$ and returns `false` otherwise.
$lowerThanOrEqualTo(o_1, o_2)$	Returns `true` if $o_1 \leq o_2$ and returns `false` otherwise.
$setUndefined(o)$	Sets o to `null`.

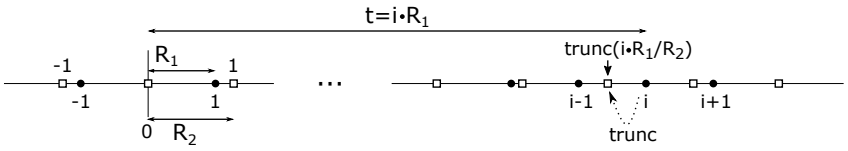
Table 4.1: Description of primitive common mappings.

$equal(Numeric\ n_1,\ Numeric\ n_2)$: returns the *Boolean* value `true` iff n_1 and n_2 represent the same *Numeric* value. If n_1 and n_2 have different *Numeric* data types an *implicit* casting is applied before the comparison following the next rules.

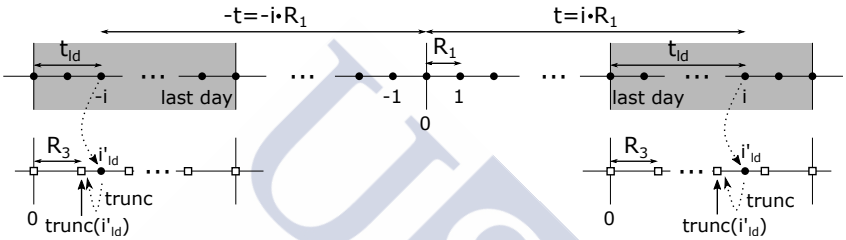
- if one argument (n_r) is a *Real* value and the other argument (n_i) is an *Integer* value, then n_r is cast to an *Integer* value yielding $trunc(n_r)$.
- if one argument (n_r) is a *Real* value and the other argument (n_{fp}) is a *FixedPrecision(P,S)* value, then n_{fp} is cast to a *Real* value.
- if one argument (n_i) is an *Integer* value and the other argument (n_{fp}) is a *FixedPrecision(P,S)* value, then n_{fp} is cast to an *Integer* value yielding $trunc(n_{fp})$.

$equal(Temporal\ t_1,\ Temporal\ t_2)$: returns the *Boolean* value `true` iff t_1 and t_2 represent the same *Temporal* value. If t_1 and t_2 have different *Temporal* data types or have the same data type but different resolution an *implicit* casting is first applied. Notice that the casting process may introduce truncation errors. Once both arguments have the same data type and resolution, $t_1 = i_1 \cdot R$ and $t_2 = i_2 \cdot R$, the *Boolean* value `true` is returned if and only if $i_1 = i_2$. Rules for the implicit casting are detailed below.

- if both arguments have the same *Temporal* data type but different resolution (thus, *Date* data type does not apply here), then the argument with higher resolution is



(a) Casting from *TimeInstant(R_1)* to *TimeInstant(R_2)*.



(b) Casting from *TimeInstant(R_1)* to *Time(R_3)*.

Figure 4.6: Temporal type castings.

cast to the lower resolution. Fig. 4.6(a) shows the casting of a *TimeInstant(R_1)* value $t = i \cdot R_1$ to the corresponding *TimeInstant(R_2)* value. First, the index of t at resolution R_2 is calculated as

$$i' = i \cdot \frac{R_1}{R_2}$$

Since only integer indexes are allowed, the integer part of previous index is used to calculate the resulting value

$$t' = \text{trunc}(i') \cdot R_2 = \text{trunc}\left(i \cdot \frac{R_1}{R_2}\right) \cdot R_2$$

– if one argument is a *TimeInstant(R_1)* value, $t_1 = i_1 \cdot R_1$, and the other argument is a *Date* value, since *Date* is equivalent to *TimeInstant(86400)*, then we can represent the *Date* argument as $t_2 = i_2 \cdot 86400$ and apply the previous rule.

– if one argument is a *TimeInstant(R_1)* value, $t_1 = i_1 \cdot R_1$, and the other argument is a *Time(R_2)* value, then t_1 is cast to the corresponding *Time(R_3)* value, where $R_3 = \min(R_1, R_2)$. Since *Time(R_3)* denotes a time shift within a specific day, only

the elapsed time since the beginning of the *last day*[3] of the period determined by t_1 (t_{ld}) is cast to *Time(R_3)*. Fig. 4.6(b) shows the casting of t and $-t$ to *Time(R_3)*. In the first case,

$$t_{ld} = t \bmod 86400$$

in the second case,

$$t_{ld} = 86400 - (-t \bmod 86400)$$

Hence, the index of t_{ld} at resolution R_3 is

$$i'_{ld} = \frac{t_{ld}}{R_3}$$

Thus, the resulting *Time(R_3)* value is

$$t'_{ld} = \text{trunc}(i'_{ld}) \cdot R_3 = \text{trunc}\left(\frac{t_{ld}}{R_3}\right) \cdot R_3$$

– if one argument is a *Time(R_1)* value and the other argument is a *Date* value, $t_2 = i_2 \cdot 86400$, then t_2 is cast to the *Time(R_1)* value $t_1 = 0$. Since the resolution of t_2 is 86400, t_2 always match the beginning of the corresponding day, and thus, always match the initial value of the *Time(R_1)* data type.

equal(*Point1D* p_1, *Point1D* p_2): returns the *Boolean* value `true` iff p_1 and p_2 represent the same *Point1D* value. If p_1 and p_2 have different precision or resolution, an *implicit* casting is first applied. Similarly to *Temporal* values, the casting process may introduce rounding errors. Once both arguments have the same precision and resolution, $p'_1 = i'_1 \cdot R'$ and $p'_2 = i'_2 \cdot R'$, the *Boolean* value `true` is returned if and only if $i'_1 = i'_2$.

Let $p_1 = i_1 \cdot R_1$ be a *Point1D(P_1,R_1)* value and $p_2 = i_2 \cdot R_2$ be a *Point1D(P_2,R_2)* value. Both arguments are cast to *Point1D(P',R')*, where $R' = \min(R_1, R_2)$ and P' enables the representation of all the values of *Point1D(P_1,R_1)* and *Point1D(P_2,R_2)* at resolution R'. The representation of all the values of each data type[4] can be ensured by enabling the representation of the maximum positive values

[3] For negative t_1 values, *last day* actually refers to the first day of the period determined by t_1.
[4] *Point1D(P,R)* is symmetrical with respect to 0.

$$p_{1max} = (10^{P_1} - 1) \cdot R_1 \qquad p_{2max} = (10^{P_2} - 1) \cdot R_2$$

Hence, the maximum value to be represented is

$$p_{max} = \max(p_{1max}, p_{2max})$$

The maximum value that *Point1D(P′,R′)* data type can represent is

$$p_{3max} = (10^{P'} - 1) \cdot R'$$

Thus, from the condition ensuring the representation of all values, the resulting precision can be isolated

$$\begin{aligned} p_{3max} &\geq p_{max} \\ (10^{P'} - 1) \cdot R' &\geq \max(p_{1max}, p_{2max}) \\ P' &\geq \log\left(\frac{\max(p_{1max}, p_{2max})}{R'} + 1\right) \\ P' &= \left\lceil \log\left(\frac{\max(p_{1max}, p_{2max})}{R'} + 1\right) \right\rceil \end{aligned}$$

The casting process is similar to the casting of *Temporal* values explained above. The integer indexes of p_1 and p_2 at resolution R' are calculated as follows

$$i'_1 = \text{round}\left(i_1 \cdot \frac{R_1}{R'}\right) \qquad i'_2 = \text{round}\left(i_2 \cdot \frac{R_2}{R'}\right)$$

equal(*Point2D* p_1, *Point2D* p_2): returns the *Boolean* value `true` iff p_1 and p_2 represent the same *Point2D* value. As in the *Point1D* case, an *implicit* error-prone casting has to be done before the comparison if p_1 and p_2 have different precision or resolution. Once both arguments have the same precision and resolution, $p'_1 = (x'_1, y'_1) = (i'_{1x} \cdot R',\ i'_{1y} \cdot R')$ and $p'_2 = (x'_2, y'_2) = (i'_{2x} \cdot R',\ i'_{2y} \cdot R')$, the *Boolean* value `true` is returned if and only if $(i'_{1x},\ i'_{1y}) = (i'_{2x},\ i'_{2y})$.

Let $p_1 = (x_1, y_1) = (i_{1x}, i_{1y}) \cdot R_1$ be a *Point2D(P*$_1$*,R*$_1$*)* value and $p_2 = (x_2, y_2) = (i_{2x}, i_{2y}) \cdot R_2$ be a *Point2D(P*$_2$*,R*$_2$*)* value. Both arguments are cast to *Point2D(P′,R′)*. Fig. 4.7 depicts the casting from p_1 to p'_1. In this case, the representation of all the values of

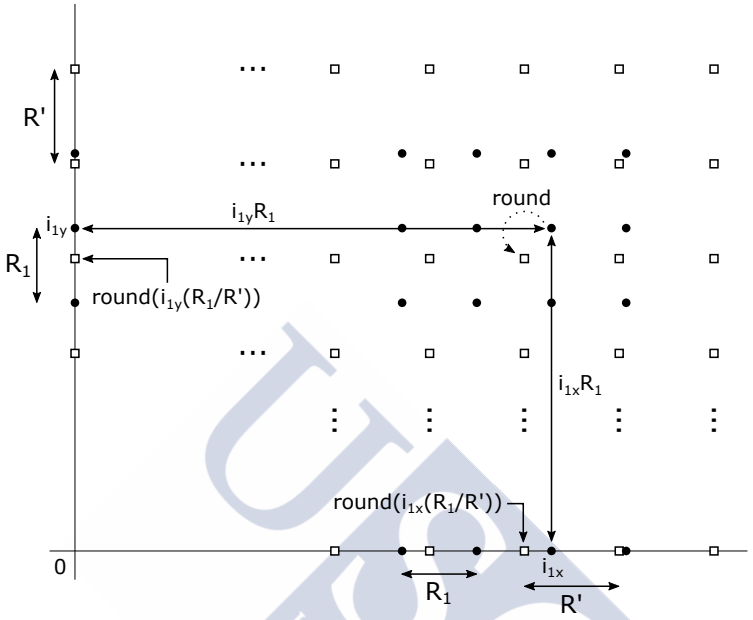


Figure 4.7: Casting from *Point2D*(P_1, R_1) to *Point2D*(P', R').

each data type[5] can be ensured by enabling the representation of the maximum positive values for each dimension,

$$p_{1x_{max}} = x_{1max} \qquad p_{2x_{max}} = x_{2max}$$
$$p_{1y_{max}} = y_{1max} \qquad p_{2y_{max}} = y_{2max}$$

Since *Point2D* has the same precision and resolution for both dimensions, the maximum positive values of both dimensions are equal, $x_{max} = y_{max} = (10^P - 1) \cdot R$. Hence, the maximum positive values for each data type, p_{1max} and p_{2max}, are calculated as follows

$$\begin{aligned} p_{1max} &= \max\left(p_{1x_{max}}, p_{1y_{max}}\right) = \max\left(x_{1max}, y_{1max}\right) \\ &= \max\left(\left(10^{P_1} - 1\right) \cdot R_1, \left(10^{P_1} - 1\right) \cdot R_1\right) = \left(10^{P_1} - 1\right) \cdot R_1 \end{aligned}$$

$$\begin{aligned} p_{2max} &= \max\left(p_{2x_{max}}, p_{2y_{max}}\right) = \max\left(x_{2max}, y_{2max}\right) \\ &= \max\left(\left(10^{P_2} - 1\right) \cdot R_2, \left(10^{P_2} - 1\right) \cdot R_2\right) = \left(10^{P_2} - 1\right) \cdot R_2 \end{aligned}$$

[5] *Point2D(P,R)* is symmetrical with respect to $(0,0)$.

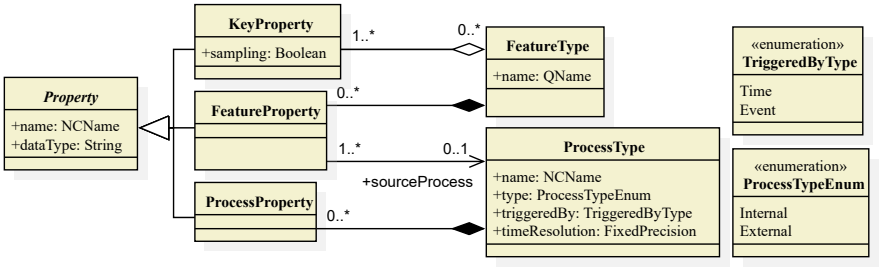


Figure 4.8: Observation Data Model (UML class diagram)

Thus, the resulting precision can be expressed again as

$$P' = \left\lceil \log\left(\frac{\max(p_{1max}, p_{2max})}{R'} + 1\right)\right\rceil$$

And the integer indexes of p_1 and p_2 are

$$i'_{1x} = \text{round}\left(i_{1x} \cdot \frac{R_1}{R'}\right) \qquad i'_{2x} = \text{round}\left(i_{2x} \cdot \frac{R_2}{R'}\right)$$
$$i'_{1y} = \text{round}\left(i_{1y} \cdot \frac{R_1}{R'}\right) \qquad i'_{2y} = \text{round}\left(i_{2y} \cdot \frac{R_2}{R'}\right)$$

equal(*Geometry* g_1, *Geometry* g_2): returns the *Boolean* value `true` iff g_1 and g_2 are *spatially equal* as defined by `ST_Equals()` method in [58]. It is a shortcut for *isEmpty*(*symDifference*(g_1,g_2)).

4.2.2 Observation Data Model

As already stated in Section 1.4, one of the main goals of SODA is to provide the users with the *observation* semantics that allow them to process and analyze observation spatio-temporal data. To achieve this fundamental objective, a novel data model providing the required *observation* semantics is defined.

The UML diagram of the proposed model is depicted in Fig. 4.8. Such model relies in the concept of *Feature* to enable the modeling of *samplings* and *observed entities*. *Feature Type* enables the classification of *Features*. Each *Feature Type* has one or more *Key Properties*, and may have additional *Feature Properties*. Both key properties of *entities* and dimensions of *samplings* can be modeled using *Key Properties*. Notice that a *Key Property* may either exist

as an independent object within the system (i.e., not aggregated to any *Feature Type*) or be aggregated to more than one *Feature Type*. Each *Feature Property* is used to model either a *sampled* property or a non-key property of an *entity*, and may be observed by a *sourceProcess*.

SourceProcesses are classified into *Process Types* which enable the characterization of some important parameters such as process type or trigger type. Regarding the former, processes may be classified either as *Time-triggered* or *Event-triggered*. The latter enables the declaration of processes either as *External* or *Internal*[6]. The execution of an *External* process is performed outside the system, thus produced observations have to be loaded into the data warehouse running typical ETL tasks. In contrast to *External* processes, the execution of an *Internal* process is performed within the system during the ETL tasks to produce new observations derived from both the imported data and the observations already stored in the data warehouse.

The schema of a spatial observation data warehouse following the data model depicted in Fig. 4.8 can be easily specified using a novel XML-based language, called XODDL (XML Observation Data Definition Language). Indeed, the definition of XODDL is deeply rooted in the observation data model, as shown in the XML schema definition of XODDL depicted in Code 4.1.

It is necessary for system administrators to perform the following tasks when starting the system up:

1. Create a dataset schema using XODDL.
2. Add *Observation Process* metadata.
3. Define *Internal Processes*.
4. Insert *Feature Types*.
5. Insert *FeatureProperties*.

Then, observation data ETL tasks can be executed within the system. New observed values, stamped with appropriate temporal data (*phenomenon Time*), are appended to *Observed Properties* of *Features Types*. Except for administration purposes, data deletions and updates are not supported.

[6] Definition of *Internal* processes is covered in Section 4.3.2

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault ="unqualified" elementFormDefault ="qualified"
   targetNamespace="es.usc.citius.de.soda.xoddl"
   version="1.0.0" xmlns="es.usc.citius.de.soda.xoddl"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="ObservationSchema">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ProcessType" type="ProcessType_Type" maxOccurs="unbounded"/>
        <xs:element name="FeatureType" type="FeatureType_Type" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="FeatureType_Type" >
    <xs:sequence>
      <xs:element name="KeyProperty" type="KeyProperty_Type" maxOccurs="unbounded"/>
      <xs:element name="Property" type="FeatureProperty_Type" minOccurs="0"
                  maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:QName" use="required"/>
  </xs:complexType>

  <xs:complexType name="ProcessType_Type" >
    <xs:sequence>
      <xs:element name="Property" type="ProcessPropertyType" minOccurs="0"
                  maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:QName" use="required"/>
    <xs:attribute name="type" type="ProcessTypeEnum" use="required"/>
    <xs:attribute name="triggeredBy" type="TriggeredByType" use="required"/>
    <xs:attribute name="timeResolution" type="xs:string" use="optional"/>
  </xs:complexType>

  <xs:simpleType name="ProcessTypeEnum">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Internal"/>
      <xs:enumeration value="External"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="TriggeredByType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Time"/>
      <xs:enumeration value="Event"/>
    </xs:restriction>
  </xs:simpleType>
```

```
  <xs:complexType name="KeyPropertyType">
    <xs:attribute name="name" type="xs:NCName" use="optional"/>
    <xs:attribute name="type" type="xs:string" use="optional"/>
    <xs:attribute name="sampling" type="xs:boolean" use="optional" default="false"/>
  </xs:complexType>

  <xs:complexType name="FeaturePropertyType">
    <xs:attribute name="name" type="xs:NCName" use="required"/>
    <xs:attribute name="type" type="xs:string" use="required"/>
    <xs:attribute name="sourceProcessType" type="xs:QName" use="optional"/>
  </xs:complexType>

  <xs:complexType name="ProcessPropertyType">
    <xs:attribute name="name" type="xs:NCName" use="required"/>
    <xs:attribute name="type" type="xs:string" use="required"/>
  </xs:complexType>
</xs:schema>
```

Code 4.1: XML schema definition of XODDL.

A more detailed description of main elements in the observation data model, *Feature Type* and *Process Type* (and associated *Dimensions* and *ExtensionalMappingSets* generated to store their data), are provided below. The UML object diagram of Fig. 4.9 shows a running example used to ease the understanding of the above concepts.

Feature Type

Feature Type has been defined to enable the integrated modeling of *entities* and *samplings*. In the running example, three *Feature Types* have been defined.

`Topo`: used to model a geographic *sampling*. The spatial *Dimension* of `Topo` is modeled by *Key Property* `Loc5m` which defines a *sampling Dimension* of data type *Point2D(9,5)*. The elevation above the sea level at each point of `Loc5m` is provided by *Feature Property* `Elevation`.

`Municipality`: used to model municipal *entities*. Each municipality is uniquely identified by non sampling *Key Property* `MunCode`. *Feature Properties* `Name` and `Geo` provide the name and geometry of each municipality respectively.

`Station`: models meteorological facilities (*entities*). Similarly to `Municipality`, a *Key Property* `StationId` uniquely identifies each meteorological station. Non observed

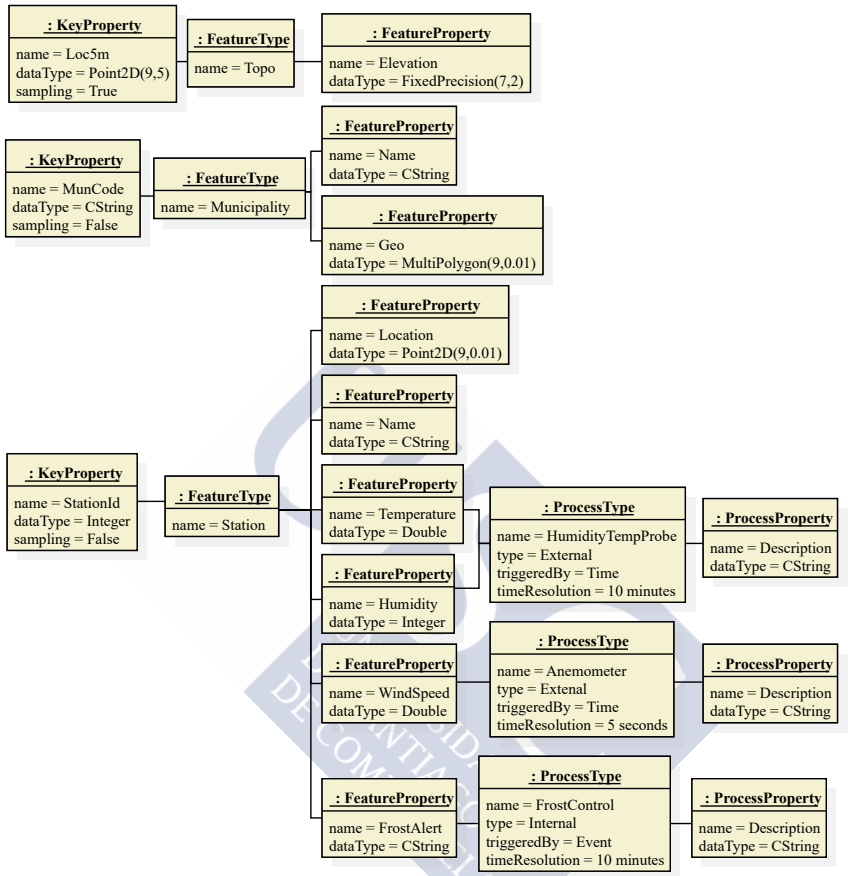


Figure 4.9: Running example (UML object diagram)

properties `Name` and `Location` provide the name and location of each station. *Observed properties* `Temperature`, `Humidity`, `WindSpeed` and `FrostAlert` model *observed values* provided by relevant *observation processes*. Temperature and relative humidity observations are provided by an external process of type `HumidityTempProbe`, wind speed observations are provided by an external process of type `Anemometer` and the frost risk index for each weather station is generated by an *internal* process of type `FrostControl` according to temperature and relative humidity observation values provided by an *external* process of type `HumidityTempProbe`.

A detailed description of *Dimensions* and *Extensional MappingSets* generated within the system for each *Feature Type* FT is provided below.

a) Let KP be a *Key Property* of data type DT. Different *Dimensions* will be generated depending on the value of attribute `sampling`.

   - if `sampling=true`, a sampling *Dimension* $FT.KP(lo,hi)$ is generated, where lo and hi of data type DT define the boundaries of the generated sampling.
   - if `sampling=false`, a non sampling *Dimension* $FT.KP : DT$ is generated.

   The running example generates the following *Dimensions*:

```
Topo.Loc5m(lo:Point2D(9,5), hi:Point2D(9,5))
Municipality.MunCode:CString
Station.StationId:Integer
```

   Notice that duplicate names are not allowed, thus the name of the *Feature Type* is added as a prefix to the name of the relevant property in order to avoid name conflicts.

b) Let $KP_1,\ldots,KP_n$ be the *Key Properties* of FT. The following *Extensional MappingSet* is also generated

$$FT(FT.KP_1,\ldots,FT.KP_n \mid M_1 : DT_1,\ldots,M_m : DT_m)$$

   where

$$M_i : DT_i = FT.FP_i(FT.KP_1,\ldots,FT.KP_n) : DT_i$$

   is the *Extensional Mapping* generated for the non observed *Feature Property* FP_i of FT.

   The *Extensional MappingSets* generated in the running example are the following:

```
Topo(
    Topo.Loc5m |
    Elevation:FixedPrecision(7,2)
Municipality(
    Municipality.MunCode |
    Geo:MultiPolygon(9,0.01))
Station(
    Station.StationId |
    Name:CString,
    Location:Point2D(9,0.01))
```

Similarly to *Key Properties*, the name of the *Feature Type* is added as a prefix to the name of the relevant *Extensional Mapping* to avoid name conflicts.

c) Let $FP_1 : DT_1, \ldots, FP_n : DT_n$ be the *Feature Properties* generated by an observation source process of type PT. Let $KP_1, \ldots, KP_n$ be the *Key Properties* of FT. The following *Extensional MappingSet* is stored to enable the recording of generated *observation values*.

$$FT.PT(FT.KP_1, \ldots, FT.KP_n, PT.Time \mid FP_1 : DT_1, \ldots, FP_n : DT_n, Process : Integer)$$

where

$$FP_i : DT_i = FP_i(FT.KP_1, \ldots, FT.KP_n, PT.Time) : DT_i$$

is the *Extensional Mapping* recording the observation values of FP_i,

$$Process(FT.KP_1, \ldots, FT.KP_n, PT.Time) : Integer$$

records the identifier of the specific observation process used at each time instant to generate observations, and

$$PT.Time$$

is a *Dimension* generated by PT to store the time instants of *observation values*.

In the running example the following *Extensional MappingSets* are generated:

```
Station.HumidityTempProbe(
    Station.StationId,
    HumidityTempProbe.Time |
    Temperature:Double,
    Humidity:Integer,
    Process:Integer)
Station.Anemometer(
    Station.StationId,
    Anemometer.Time |
    WindSpeed:Double,
    Process:Integer)
Station.FrostControl(
    Station.StationId,
    FrostControl.Time |
    FrostAlert:CString,
    Process:Integer)
```

Process Type

Source Processes metadata of *Observed Properties* are stored by *Process Type* objects. Each *Process Type* may be either *Time-triggered* or *Event-triggered* at a given `timeResolution` *R*. *Time-triggered* processes generate *Temporal samplings* at resolution *R*, i.e., a new *observed value* is generated every *R* seconds since *lo* to *hi* (*lo* and *hi* are respectively the lowest and highest *TimeInstant(R)* values defined in the system for observation times generated by the relevant process). The semantics for `timeResolution` in *Event-triggered* processes is slightly different, meaning that the time at which the event is fired will be stored as a *TimenInstant(R)* value. In the running example, process `HumidityTempProbe` generates temperature and humidity observations every 10 minutes, whereas process `Anemometer` generates wind speed observations every 5 seconds. Owing to the *external* nature of these processes, *observation values* must be provided by *external* systems. On the contrary, *internal* `FrostControl` process computes the frost risk value according to meteorological observations provided by `HumidityTempProbe`. To generate *calculated Feature Properties*, the system executes *internal* processes during ETL tasks.

For each *Process Type PT*, the following *Dimensions* and *Extensional MappingSets* are recorded.

a) Since the *sourceProcess* that actually generates *observation* values for *PT* may change over time, identifiers of such processes are automatically generated by the system and recorded in a *Dimension PT* : *Integer*. Notice that these identifiers are used in *Extensional Mappings Process*, defined in the above subsection, to identify the *sourceProcess* that generates each observation. Following *Dimensions* are generated within the system to record all the required process identifiers in the running example:

```
HumidityTempProbe:Integer
Anemometer:Integer
FrostControl:Integer
```

b) Let $PP_1, \ldots, PP_n$ be the *Process Properties* of *PT*. The following *Extensional MappingSet* is recorded

$$PT.Properties(d_{PT} \mid M_1 : PPT_1, \ldots, M_n : PPT_n)$$

where

$$d_{PT} = PT : Integer$$

is the *Dimension* of PT identifiers, and

$$M_i : PPT_i = PT.PP_i(PT) : PPT_i$$

is the *Extensional Mapping* that enables the recording of the PP_i values for each PT instance.

For the running example, the following *Extensional MappingSets* are generated:

```
HumidityTempProbe.Properties(
    HumidityTempProbe |
    Description:CString)
Anemometer.Properties(
    Anemometer |
    Description:CString)
FrostControl.Properties(
    FrostControl |
    Description:CString)
```

c) If PT is a *time-triggered* Process of resolution R then a *Sampling*

$$PT.Time(lo : TimeInstant(R), hi : TimeInstant(R))$$

is recorded, where lo and hi are respectively the lowest and highest time instants configured in SODA for observations generated by PT. If PT is an *event-triggered* Process of resolution R then a non-sampling *Dimension*

$$PT.Time : TimeInstant$$

is stored. These *Dimensions* store the time instant assigned to each *observation value* and are, therefore, added to the domain of the relevant *Extensional MappingSet* that stores such *observation values*. As shown in the above subsection, *Dimensions* `HumidityTempProbe.Time`, `Anemometer.Time` and `FrostControl.Time` have been added to the domain of *Extensional MappingSets* `Station.HumidityTempProbe`, `Station.Anemometer` and `Station.FrostControl`, respectively.

4.3 Observation Data Analysis

Given the above data models for the representation of both spatio-temporal and observation data, a novel XML based language, called MAPAL (Mapping Analysis Language) is provided

for the analysis of the proposed data structures. Such a language should fulfill the following requirements based on the generic functionality of an observation data management system.

- Follow a declarative paradigm.
- Support for OLAP over large data warehouses of spatial observation data.
- Support the definition of *Internal* Processes.
- Support the integrated analysis of both *entity* data and spatial, temporal and spatio-temporal *sampled* data.
- Support for aggregation functionality.

4.3.1 Mapping Analysis Language (MAPAL)

Owing to the functional nature of the proposed data models, extensions of well known languages like SQL and XQuery cannot be directly used. However, constructs of these well known languages are used by the hybrid logical-functional paradigm of MAPAL. The combination of such constructs with the XML syntax enables their insertion in currently dominating web services interfaces. Three types of expressions (*Functional*, *Conditional* and *Aggregate*) may be used to define derived *Constants*, *Intensional Mappings* and *Extensional MappingSets*. Additionally, *Sampling* and *Dimension* expressions are used to define derived *Dimensions*. MAPAL syntax and semantics, with illustrative examples, are provided below.

The XML Schema definition of MAPAL is shown in Code 4.2. Notice that, for illustration purposes, pieces of code defining *Dimensions*, *Intensional Mappings*, *Constants* and *Extensional MappingSets* are explained separately and corresponding references have been inserted in Code 4.2.

An abstract super type *DefinitionType* is defined to encapsulate the required common attribute `name`. As it is shown in following code snippets, all definitions extend *DefinitionType*. *ExternalReferenceType* specifies the syntax required to access input and output data channels in order to import and export *Constants*, *Dimensions* and *Extensional MappingSets*. Two required attributes, `dataChannel` and `name`, have been defined to specify the data channel and the name of the *Constant*, *Dimension* or *Extensional MappingSet* in the data channel, respectively. Code 4.3 shows the *DimensionType* schema that enables the definition of *sampling* and non sampling *Dimensions*. *ConstantType* schema that enables the definition of *Constants* is depicted in Code 4.4. In Code 4.5 the *IntensionalMappingType* schema that enables the defi-

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           targetNamespace="es.usc.citius.de.mapal"
           version="1.0.0"
           xmlns="es.usc.citius.de.mapal"
           xmlns:xs="http://www.w3.org/2001/XMLSchema" >

<!-- ***************************** -->
<!-- DEFINITION                    -->
<!-- ***************************** -->
<xs:complexType abstract="true" name="DefinitionType">
  <xs:attribute name="name" type="xs:NCName" use="required"/>
</xs:complexType>
<xs:element abstract="true" name="Definition" type="DefinitionType"/>

<!-- ***************************** -->
<!--  EXTERNAL REFERENCES -->
<!-- ***************************** -->
<xs:complexType name="ExternalReferenceType">
  <xs:attribute name="dataChannel" type="xs:NCName" use="required"/>
  <xs:attribute name="name" type="xs:QName" use="required"/>
</xs:complexType>

<!-- DIMENSION DEFINITION:  Code 4.3           -->
<!-- INTENSIONAL MAPPING DEFINITION:  Code 4.5 -->
<!-- CONSTANT DEFINITION:  Code 4.4            -->
<!-- EXTENSIONAL MAPPING DEFINITION:  Code 4.6 -->
</xs:schema>
```

Code 4.2: XML schema definition of MAPAL.

nition of *Intensional Mappings* is shown. Code 4.6 shows the *ExtensionalMappingSetType* schema that enables the definition of *Extensional MappingSets*.

Dimensions

DimensionType, Code 4.3, specifies the syntax to define *Dimensions*. Such *Dimensions* may be defined by using the element `<Dimension>`. *DimensionType* extends *DefinitionType* with an optional attribute `storeName`. If this attribute is defined, the *Dimension* is persisted to disk with the name `storeName` and added to the system catalog in order to be accessible for subsequent operations. Notice that the common attribute `name` refers to the name of the *Dimension* in main memory. The new generated *Dimension* may be exported to a specific number of data channels by adding one optional element `<Output>` of type *ExternalReferenceType* for each

```
<xs:group name="DimensionSpecification">
  <xs:sequence>
    <xs:element name="ForEach" type="ForEachType" maxOccurs="unbounded"/>
    <xs:element name="Where" type="xs:string" minOccurs="0"/>
    <xs:element name="Return" type="xs:string"/>
  </xs:sequence>
</xs:group>

<xs:complexType name="ForEachType">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="var" type="xs:NCName" use="required"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="DimensionType">
  <xs:complexContent>
    <xs:extension base="DefinitionType">
      <xs:sequence>
        <xs:choice>
          <xs:element name="Input" type="ExternalReferenceType"/>
          <xs:element name="Sampling" type="SamplingSpecificationType"/>
          <xs:group ref="DimensionSpecification" />
        </xs:choice>
      </xs:sequence>
      <xs:attribute name="storeName" type="xs:QName" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="Dimension" substitutionGroup="Definition" type="DimensionType"/>

<xs:complexType name="SamplingSpecificationType">
  <xs:sequence>
    <xs:element name="Start" type="xs:string"/>
    <xs:element name="End" type="xs:string"/>
  </xs:sequence>
  <xs:attribute name="type" type="xs:string"/>
</xs:complexType>
```

Code 4.3: XML schema definition of MAPAL Dimension.

output data channel. For the `<Output>` element, `dataChannel` is the data channel to which the *Dimension* is exported and `name` is the storage name of the *Dimension* in the data channel.

As already stated, a *Dimension* may be either a *sampling Dimension* or a non sampling *Dimension*. Moreover, each non sampling *Dimension* may be either derived from *Dimensions* previously loaded in the system or loaded from *external* data channels.

A *sampling Dimension* may be defined by using the element `<Sampling>` of type *SamplingSpeficicationType* which specifies the appropriate syntax to define *sampling Dimensions*. An attribute `type` identifying the *sampling Dimension* data type is required. Elements `<Start>` and `<End>` enable the definition of the minimum and maximum values of *sampling Dimensions*. An example of a *Point2D sampling Dimension* covering the geographic region of Galicia (autonomous community of northwestern Spain) is shown next.

```
<Dimension name="Galicia100m" storeName="local:Galicia100m">
  <Sampling type="Point2D(9,100)">
    <Start> 465000.0,4615000.0 </Start>
    <End> 705000.0,4865000.0 </End>
  </Sampling>
</Dimension>
```

A derived non sampling *Dimension* may be defined by using a sequence of elements `<ForEach>`, `<Where>` and `<Result>`. The element `<ForEach>` contains a *Dimension Set Expression* of the form

$$d_1\ OP\ d_2\ OP\ \ldots\ OP\ d_m$$

where d_i is a *Dimension* name and *OP* is either *AND* or *OR*. The semantics of *AND* and *OR* are respectively those of *Dimension Set Operations Intersection* and *Union*, which are defined below. The required attribute `var` defines a variable name that iterates over the result of the *Dimension Set Expression*. Multiple elements `<ForEach>` may be defined inside the element `<Dimension>` enabling the definition of multiple variables iterating over different *Dimension Set Expressions*. If multiple variables are defined, the values to be processed are provided by the Cartesian product of the values resulting from each *Dimension Set Expression*. Then, values to be processed may be filtered by providing a conditional expression with element `<Where>`. Finally, the resulting values of the previous condition are processed by the expression specified by the element `<Result>`. An example is shown below.

```
<Dimension name="Pontevedra100m" storeName="local:Pontevedra100m">
  <ForEach var="x"> galicia100m </ForEach>
  <Where> getX(x) &gt; 490000 AND getX(x) &lt; 600000 AND
          getY(x) &gt; 4630000 AND getY(x) &lt; 4750000
  </Where>
  <Return> x </Return>
</Dimension>
```

Dimension `Pontevedra100m` contains the *Point2D* values within `Galicia100m` which meet the condition specified in element `<Where>`, i.e., the points within a rectangular grid of *Point2D* values covering the geographic region of Pontevedra (province of Galicia). Although `Pontevedra100m` is a rectangular grid of *Point2D* values, it is not a *sampling Dimension* anymore. Obviously, `Galicia100m` must be already defined in the system. Since attribute `storeName` has been defined, a *Dimension* `local:Pontevedra100m` is stored on disk to record `Pontevedra100m` values.

Any external *Dimension* may be imported by using the element `<Input>` of type *ExternalReferenceType*. For the `<Input>` element, `dataChannel` is the data channel from which the *Dimension* is imported and `name` specifies the name in the input data channel of the *Dimension* to be imported. The following example imports *Dimension* `Galicia5m` from *Dimension* `tiff:GaliciaLoc5m` defined in data channel `GeoTiff` and stores it into local catalog.

```
<Dimension name="Galicia5m" storeName="local:Galicia5m">
  <Input dataChannel="GeoTiff" name="tiff:GaliciaLoc5m"/>
</Dimension>
```

Let $d_1(T_1)$ and $d_2(T_2)$ be two non sampling *Dimensions*, where T_1 and T_2 are compatible data types, i.e., either they are of the same data type or an *implicit* casting has been defined among them. Then d_1 *Union* d_2 is defined as the following *Dimension*

$$d(T) = \text{cast}(d_1 \; as \; T) \cup \text{cast}(d_2 \; as \; T)$$

where T is the data type resulting from the *implicit* casting between T_1 and T_2, and $\text{cast}(d_i asT)$ is the *Dimension* resulting from casting each element of d_i to type T.

Let $d_1(T_1)$ and $d_2(T_2)$ be two *Dimensions*, where T_1 and T_2 are compatible data types whose resulting *implicit* casting data type is T, and at least one of the *Dimensions* is a 1D *Sampling*. Then d_1 *Union* d_2 is defined as the 1D *Sampling* $S(m,M)$, where

$$m = \min\{\text{cast}(v \; as \; T) \mid v \in d_1 \cup d_2\}$$
$$M = \max\{\text{cast}(v \; as \; T) \mid v \in d_1 \cup d_2\}$$

Let $d_1(T_1)$ and $d_2(T_2)$ be two *Dimensions*, where T_1 and T_2 are compatible data types whose resulting *implicit* casting data type is T, and at least one of the *Dimensions* is a 2D *Sampling*. Then d_1 *Union* d_2 is defined as the 2D *Sampling* $S((x_m,y_m),(x_M,y_M))$ where

$$x_m = \min\{\text{cast}(x \text{ as } T) \mid (x,y) \in d_1 \cup d_2\}$$
$$y_m = \min\{\text{cast}(y \text{ as } T) \mid (x,y) \in d_1 \cup d_2\}$$
$$x_M = \max\{\text{cast}(x \text{ as } T) \mid (x,y) \in d_1 \cup d_2\}$$
$$x_M = \max\{\text{cast}(y \text{ as } T) \mid (x,y) \in d_1 \cup d_2\}$$

Let $d_1(T_1)$ and $d_2(t_2)$ be two *Dimensions*, where T_1 and T_2 are compatible data types whose resulting *implicit* casting type is T, and at least one of the *Dimensions* is a non sampling *Dimension*. Then d_1 *Intersection* d_2 is defined as the following *Dimension*,

$$d(T) = \text{cast}(d_1 \text{ as } T) \cap \text{cast}(d_2 \text{ as } T)$$

where $\text{cast}(d_i \text{ as } T)$ is the *Dimension* resulting from casting each element of d_i to type T.

Let $d_1(T_1)$ and $d_2(T_2)$ be two 1D *Samplings*, where T_1 and T_2 are compatible data types whose resulting *implicit* casting data type is T. Then d_1 *Intersection* d_2 is defined as the 1D *Sampling* $S(m,M)$, where

$$m = \max\{\text{cast}(v \text{ as } T) \mid v \in d_1 \cup d_2\}$$
$$M = \min\{\text{cast}(v \text{ as } T) \mid v \in d_1 \cup d_2\}$$

Let $d_1(T_1)$ and $d_2(T_2)$ be two 2D *Samplings*, where T_1 and T_2 are compatible data types whose resulting *implicit* casting data type is T. Then d_1 *Intersection*;d_2 is defined as the 2D *Sampling* $S((x_m,y_m),(x_M,y_M))$ where

$$x_m = \max\{\text{cast}(x \text{ as } T) \mid (x,y) \in d_1 \cup d_2\}$$
$$y_m = \max\{\text{cast}(y \text{ as } T) \mid (x,y) \in d_1 \cup d_2\}$$
$$x_M = \min\{\text{cast}(x \text{ as } T) \mid (x,y) \in d_1 \cup d_2\}$$
$$x_M = \min\{\text{cast}(y \text{ as } T) \mid (x,y) \in d_1 \cup d_2\}$$

Constants

The syntax to define *Constants* is specified by `ConstantType` in Code 4.4. `storeName`, `<Input>` and `<Output>` have been defined with exactly the same semantics that those defined for *Dimensions*. Additionally, a *Constant* may be defined through a *Functional Expression* by using the element `<Return>`.

```
<xs:complexType name="ConstantType">
  <xs:complexContent>
    <xs:extension base="DefinitionType">
      <xs:sequence>
        <xs:choice>
          <xs:element name="Input" type="ExternalReferenceType"/>
          <xs:element name="Return" type="xs:string"/>
        </xs:choice>
        <xs:element name="Output" type="ExternalReferenceType"
                    minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="storeName" type="xs:QName" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="Constant" substitutionGroup="Definition" type="ConstantType"/>
```

Code 4.4: XML schema definition of MAPAL Constant.

A *Functional Expression e*, denoted by $e(v_1,\ldots,v_n)$, defined in the context of variables $v_1,\ldots,v_n$ combines context variables with already defined constructors (*Constants*, *Intensional Mappings* and *Extensional MappingSets*), operators, literals, *primitive* mappings and type castings. The semantics is the obvious one.

A new *Constant* `IDWDistance` is defined by the following code through a simple functional expression and then exported to a PostGIS channel. Notice that constants `dist1` and `dist2` must be already defined in the system.

```
<Constant name="IDWDistance" storeName="local:IDWDistance">
  <Result> min(dist1,dist2) </Result>
  <Output dataChannel="PostGIS" name="postgis:IDWDistance"/>
</Dimension>
```

Intensional Mappings

The schema of an *Intensional Mapping* in MAPAL is shown in Code 4.5. Definition of *Intensional Mappings* are enabled by element `<IntensionalMapping>` of type *IntensionalMappingType*. An optional attribute `domain` has been defined to enable the specification of input arguments. Thus a list of input arguments may be specified in order to define the domain over which the mapping has to be applied. The resulting *Intensional Mapping* may be defined by *Functional*, *Conditional* or *Aggregated Expressions* applied to the input argument values.

```
<xs:group name="AggregateExpression">
  <xs:sequence>
    <xs:element name="ForEach" type="ForEachType" maxOccurs="unbounded"/>
    <xs:element name="Where" type="xs:string" minOccurs="0"/>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="OrderAscendingBy" type="xs:string"/>
      <xs:element name="OrderDescendingBy" type="xs:string"/>
    </xs:choice>
    <xs:element name="Aggregate" type="xs:string"/>
  </xs:sequence>
</xs:group>

<xs:group name="ConditionalExpression">
  <xs:sequence>
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="When" type="xs:string"/>
      <xs:element name="ThenReturn" type="xs:string"/>
    </xs:sequence>
    <xs:element name="ElseReturn" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:group>

<xs:complexType name="IntensionalMappingType">
  <xs:complexContent>
    <xs:extension base="DefinitionType">
      <xs:choice>
        <xs:element name="Return" type="xs:string"/>
        <xs:group ref="AggregateExpression" />
        <xs:group ref="ConditionalExpression" />
      </xs:choice>
      <xs:attribute name="domain" type="xs:string" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="IntensionalMapping"
            substitutionGroup="Definition"
            type="IntensionalMappingType"/>
```

Code 4.5: XML schema definition of MAPAL Intensional Mapping.

Similarly to *Constants*, element `<Return>` enables the definition of an *Intensional Mapping* through a *Functional Expression*. The following example defines an *Intensional Mapping* `slope` that computes the slope of the terrain at location `p` (input argument) from its elevation. Notice that referenced *Extensional MappingSet* `Topo:Elevation` must be already defined.

```
<IntensionalMapping name="xslope" domain="p">
  <Return>
    1*Topo.Elevation(shift(p,-1,1)) +
    2*Topo.Elevation(shift(p,-1,0)) +
    1*Topo.Elevation(shift(p,-1,-1)) -
    1*Topo.Elevation(shift(p,1,1)) -
    2*Topo.Elevation(shift(p,1,0)) -
    1*Topo.Elevation(shift(p,-1,-1))
  </Return>
</IntensionalMapping>

<IntensionalMapping name="yslope" domain="p">
  <Return>
    1*Topo.Elevation(shift(p,-1,-1)) +
    2*Topo.Elevation(shift(p,0,-1)) +
    1*Topo.Elevation(shift(p,1,-1)) -
    1*Topo.Elevation(shift(p,-1,1)) -
    2*Topo.Elevation(shift(p,0,1)) -
    1*Topo.Elevation(shift(p,1,1))
  </Return>
</IntensionalMapping>

<IntensionalMapping name="slope" domain="p">
  <Return> atan1(sqrt(xslope(p)^2 + ysolpe(p)^2)) </Return>
</IntensionalMapping>
```

An *Aggregate Expression*, that enables the application of aggregate mappings to finite sequences of tuples built from *Dimensions* and *Extensional MappingSets*, may be used in MAPAL to define an *Intensional Mapping*. Several required and optional elements may be used to define an *Aggregate Expression*. Elements `<ForEach>` and `<Where>` have the same semantics that those defined by *DimensionType*. An optional sequence of *Functional Expressions* may be specified to order, either ascending or descending, the resulting values from `<ForEach>` and `<Where>` elements. Element `<Aggregate>` specifies an *Aggregate Expression* which combines functional expression elements with system provided aggregate functions. Aggregated functions defined within the system may have the form of the classical ones of SQL (e.g., AVG, SUM) or may also exploit the ordering, e.g., function $ATPOSITION(S,n)$ yields the element of S located at position n.

The following lines of code apply the well known *Inverse Distance Weight* (IDW) interpolation method over meteorological stations defined in *Dimension* `StationId` for the specific input meteorological property `m`, location point `p` and time instant `t` to provided an aggregated value of `m`.

```
<IntensionalMapping name="meteoProperty" domain="m, s, t">
  <When> m = "Temperature" </When>
  <ThenReturn> ffr:Observation.Temperature(s,t) </ThenReturn>
  <When> m = "Humidity" </When>
  <ThenReturn> ffr:Observation.Humidity(s,t) </ThenReturn>
  <When> m = "WindSpeed" </When>
  <ThenReturn> ffr:Observation.WindSpeed(s,t) </ThenReturn>
</IntensionalMapping>

<IntensionalMapping name="IDW" domain="m,p,t">
  <ForEach var = "s"> StationId </ForEach>
  <Where> distance(Station.Loc(s), p) &lt; IDWDistance </Where>
  <Aggregate> sum(meteoProperty(m,s,t)/distance(Station.Loc(s), p)^2) /
              sum(1/distance(Station.Loc(s), p)^2)
  </Aggregate>
</IntensionalMapping>
```

An *Intensional Mapping* may be defined by a *Conditional Expression* that enables the introduction of *if-then-else* structures. An unbounded number of `<When>` and `<ThenReturn>` elements enable the definition of conditions and corresponding results. An optional element `<ElseReturn>` enables the definition of the default result. In previous code, *Intensional Mapping* `meteoProperty` returns temperature, humidity or wind speed observations through a *Conditional Expression*.

Extensional MappingSets

The schema definition of an *Extensional MappingSet* of type *ExtensionalMappingSetType* is shown in Code 4.6. Similarly to *Dimensions* and *Constants*, an *Extensional MappingSet* may be imported from an external data channel and exported to several external data channels. Of course, it may also be persisted into local catalog. The following code shows the definition of *Extensional MappingSet* `Observation` imported from *Extensional MappingSet* `Observation` in data channel `Postgis`, persisted to local catalog as `ffr:Observation` and exported to data channel `NetCDF` as `ObservationFromPostgis`.

```
<ExtensionalMappingSet name="Observation" storeName="ffr:Observation">
  <Input dataChannel="Postgis" name="Observation"/>
  <Output dataChannel="NetCDF" name="ObservationFromPostgis"/>
</ExtensionalMappingSet>
```

An optional attribute `domain` may be used to specify a list of *Dimensions* so that the Cartesian product of such *Dimensions* is the domain of the resulting *Extensional MappingSet*.

```
<xs:complexType name="ExtensionalMappingType">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="name" type="xs:NCName" use="required"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="ExtensionalMappingSetType">
  <xs:complexContent>
    <xs:extension base="DefinitionType">
      <xs:sequence>
        <xs:choice>
          <xs:element name="Input" type="ExternalReferenceType"/>
          <xs:element name="ExtensionalMapping" type="ExtensionalMappingType"
                      minOccurs="0" maxOccurs="unbounded"/>
        </xs:choice>
        <xs:element name="Output" type="ExternalReferenceType"
                    minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="storeName" type="xs:QName" use="optional"/>
      <xs:attribute name="domain" type="xs:string" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="ExtensionalMappingSet" substitutionGroup="Definition"
            type="ExtensionalMappingSetType"/>
```

Code 4.6: XML schema definition of MAPAL Extensional Mapping.

Since an *Extensional MappingSet* is composed of *Extensional Mappings* that provide an output value for each domain element, `<ExtensionalMapping>` enables the definition of an *Extensional Mapping* through a *Functional Expression*. In the example below, a forest fire risk index is provided by *Extensional MappingSet* `ForestFire` for each combination of time instant (within `ffr:ObsDate`) and location (within `ffr:Loc5m`). Resulting *Extensional MappingSet* is exported to data channel `NetCDF` as `ForestFire`.

```
<IntensionalMapping name="normalize" domain="v, min, max">
  <When> v &lt; min </When> <ThenReturn>0</ThenReturn>
  <When> v &gt; max </When> <ThenReturn>1</ThenReturn>
  <ElseReturn> (v-min)/(max-min) </ElseReturn>
</IntensionalMapping>

<ExtensionalMappingSet name="ForestFire" domain="p ffr:Loc5m, t ffr:ObsDate">
  <ExtensionalMapping name="Risk">
    normalize(IDW("Temperature",p,t), minTemperature, maxTemperature)*TemperatureWeight+
```

```
    (IDW("Humidity", p,t)/100)*HumidityWeight+
    normalize(IDW("WindSpeed",p, t), minWindSpeed, maxWindSpeed)*WindSpeedWeight+
    normalize(slope(p), 0, maxSlope)*SlopeWeight
  </ExtensionalMapping>
  <Output dataChannel="NetCDF" name="ras:ForestFire"/>
</ExtensionalMappingSet>
```

4.3.2 Analytical Processes

An appropriate syntax to define *internal* analytical *Observation* processes executed during ETL tasks is now defined. Similarly to MAPAL and XODDL, a declarative XML-based syntax has been defined to ease the definition of *internal* processes.

Element `<Process>` enables the definition of such processes. A required attribute `<processType>` specifies the process data type. An optional element `<Description>` within `<Process>` may provide a textual process description. A required element `<Definition>` comprises the required MAPAL elements to properly define the *internal* process.

- First, a number of optional *Dimensions* and *Intensional Mappings* may be defined to be used in remainder elements.
- Next, the temporal *Dimension* of the resulting process is defined by using either an element `<TriggeredByTime>` or an element `<TriggeredByEvent>`. Recall that each *Process Type PT* has a *Dimension PT.Time*, which is either a *sampling Dimension* for *time-triggered* processes or a non sampling *Dimension* for *event-triggered* processes. The temporal *Dimension* of a *time-triggered internal Process Type PT* with temporal resolution *R* is defined with an expression of the following form:

  ```
  <TriggeredByTime> PT1.Time,...,PTn.Time </TriggeredByTime>
  ```

  where each $PT_i.Time$ is the temporal *Dimension* of a *Process Type* PT_i. The semantics are those of the 1D *Sampling* S(m,M), where

  $$m = \text{cast}(\min\{v \mid v \in PT_1.Time \cup PT_2.Time \cup \ldots \cup PT_n.Time\} \; as \; TimeInstant(R))$$
  $$M = \text{cast}(\max\{v \mid v \in PT_1.Time \cup PT_2.Time \cup \ldots \cup PT_n.Time\} \; as \; TimeInstant(R))$$

  An expression of the following form enables the definition of the temporal *Dimension* of an *event-triggered internal Process Type PT* of time resolution *R*:

```
<TriggeredByEvent>
  <Event var="t"> PT1.Time, PT2.Time, ... , PTn.Time </Event>
  <Condition> c(t) </Condition>
</TriggeredByEvent>
```

where each $PT_i.Time$ is the temporal *Dimension* of a *Process Type* PT_i and $c(t)$ is a functional expression of *Boolean* type. The semantics are those of the non sampling *Dimension* defined by the set

$$\{\, \text{cast}(t \; as \; TimeInstant(R)) \mid t \in PT_1.Time \cup PT_2.Time \cup \ldots \cup PT_n.Time \wedge c(t) \,\}$$

– Finally, an `<ExtensionalMapping>` MAPAL element has to be used to define a *Feature Property FT.FP* observed by an *internal Process Type PT*. This *Extensional Mapping* is automatically added to the relevant *Extensional MappingSet* recording *FP* observation values. The evaluation of each *Extensional Mapping* during ETL tasks is restricted to the evaluation of the elements of *PT.Time* to be imported, avoiding re-evaluation of the *Extensional Mapping* for the whole temporal extension of the data warehouse.

Definition of *Process Type* `FrostControl` of running example is provided below for illustration purposes.

```
<?xml version="1.0" encoding="utf-8"?>
<pd:ProcessDefinitions
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="es.usc.citius.de.soda.ProcessDefinition Soda_ProcessDefinition.xsd"
 xmlns="es.usc.citius.de.mapal"
 xmlns:fish="es.usc.citius.de.fish"
 xmlns:pd="es.usc.citius.de.soda.ProcessDefinition">

  <pd:Process processType="FrostControl">
    <pd:Description>
      Calculates the frost risk index for each Station from Temperature and Humidity
     observation values.
    </pd:Description>
    <pd:Definition>
      <IntensionalMapping name="FrostAlert" domain="t, h">
        <When> t &lt; 0 AND h &gt; 95 </When>
        <ThenReturn> VERY HIGH </ThenReturn>
        <When> t &lt; 0 AND h &gt; 85 AND h &lt;= 95 </When>
        <ThenReturn> HIGH </ThenReturn>
```

```
        <When> t &lt; 0 AND h &gt; 75 AND h &lt;= 85 </When>
        <ThenReturn> MEDIUM </ThenReturn>
        <When> t &lt; 0 AND h &gt; 65 AND h &lt;= 75 </When>
        <ThenReturn> LOW </ThenReturn>
        <When> t &lt; 0 AND h &gt; 55 AND h &lt;= 65 </When>
        <ThenReturn> VERY LOW </ThenReturn>
        <ElseReturn> VERY LOW </ElseReturn>
      </IntensionalMapping>

      <IntensionalMapping name="StationsInRisk" domain="t">
        <ForEach var="s"> Station.StationId </ForEach>
        <Where> Station.HumidityTempProbe.Temperature(s, t) &lt; 0
                AND Station.HumidityTempProbe.Humidity(s, t) &gt; 85
        </Where>
        <Aggregate> not EMPTY(s) </Aggregate>
      </IntensionalMapping>

      <pd:TriggeredByEvent>
        <pd:Event var="t"> HumidityTempProbe.Time </pd:Event>
        <pd:Condition> StationsInRisk(t)</pd:Condition>
      </pd:TriggeredByEvent>

      <ExtensionalMapping name="FrostAlert"
                          domain="Station.StationId s, FrostControl.Time t">
        <Return> FrostAlert(Station.HumidityTempProbe.Temperature(s, t),
                            Station.HumidityTempProbe.Humidity(s, t))
        </Return>
      </ExtensionalMapping>
    </pd:Definition>
  </pd:Process>
</pd:ProcessDefinitions>
```

An accurate implementation of a frost risk alert system is out of the scope of this Thesis, thus some simplifications are made for an easy understanding. For this example, the frost risk is calculated based on the following rules:

- if $T < 0 \land RH > 95$, then frost risk is VERY HIGH
- if $T < 0 \land 85 < RH \leq 95$, then frost risk is HIGH
- if $T < 0 \land 75 < RH \leq 85$, then frost risk is MEDIUM
- if $T < 0 \land 65 < RH \leq 75$, then frost risk is LOW
- if $T < 0 \land 55 < RH \leq 65$, then frost risk is VERY LOW

where T is the measured temperature in Celsius degrees and RH is measured relative humidity in percentage units.

Every time that `FrostControl` code is executed a new process identifier is automatically generated by the system. This identifier is stored in both *Dimension* `FrostControl` and *Extensional Mapping* `Process` of *Extensional MappingSet* `Station.FrostControl`. Process `FrostControl` is defined as an *event-triggered* process that is fired every time a station is in risk VERY HIGH or HIGH, i.e., measured temperature is below 0ºC and measured relative humidity is above 85%. Such conditions are implemented by *Intensional Mapping* `StationsInRisk`. *Feature Property* `Station.FrostAlert` generates output observations applying the above rules (implemented by *Intensional Mapping* `FrostAlert`) over temperature and relative humidity values.

4.3.3 System Operators

Query processing performs the evaluation of the above MAPAL expressions. A many-sorted algebra over three different data structures (*Dimensions*, *Constants* and *Extensional MappingSets*) that enables the evaluation of MAPAL queries is defined next. Operations of this algebra are classified into three different groups according to the result structure that they produce. The general syntax of such operations is the following:

$$operatorName[paramList]\ldots[paramList](argumentList)$$

where *paramList* is a comma separated list of parameters and *argumentList* is a comma separated list of arguments.

Dimension Operators

- ***ImportDimension***$[Name][channelName][storageName]$. Imports a *Dimension* from an external data channel. Parameter *Name* is the name of the new *Dimension*. Parameter *channelName* is the name of the external data channel. Parameter *storageName* is the name, in the external data channel, of the *Dimension* to import.

- ***ScanDimension***$[Name]$. Reads a *Dimension* from local catalog. Parameter *Name* is the name of the *Dimension* to read.

- ***SamplingDimension***$[Name](k_1,k_2)$. Generates an new *in-memory sampling Dimension* with all the values of *Sampling* $S(k_1,k_2)$. Parameter *Name* is a *CString* containing the

name of the generated *Dimension*. Operands k_1 and k_2 are *Constants* with identical temporal or spatial data type, obtained from some *Constant* operator.

- ***Union***(d_1, d_2). Computes the *Union* of *Dimensions* as defined in 4.3.1. Operands d_1 and d_2 are *Dimensions* produced by some other operator.
- ***Intersection***(d_1, d_2). Computes the *Intersection* of *Dimensions* as defined in 4.3.1. Operands d_1 and d_2 are *Dimensions* produced by some other operator.
- ***ProjectDimension***$[d][c](MS)$. Generates a result *Dimension* containing all the distinct elements of parameter d where *Extensional Mapping* c has a `true` value. Operand *MS* is an *Extensional MappingSet* produced by a relevant operation. Parameter d is the name of either a *Dimension* or an *Extensional Mapping* of *MS*. Optional parameter c is the name of a *Boolean Extensional Mapping* of *MS*. Notice that if d is a *sampling Dimension* and c is provided, then the resulting *Dimension* will not be a *sampling Dimension* anymore.
- ***StoreDimension***$[Name](d)$. Saves *Dimension* d to disk using *Name* as its storage name. Parameter *Name* is a *CString* and parameter d is a *Dimension* generated by some operator. If d is a *sampling Dimension* only the metadata and its limits are written into the local catalog.
- ***ExportDimension***$[channelName][storageName](d)$. Exports the *Dimension* d to an external data channel. Parameter d is a *Dimension* generated by some operator. Parameter *channelName* is the name of the external data channel. Parameter *storageName* is the name, in the external data channel, of the new exported *Dimension*.

Extensional MappingSet Operators

- ***ImportMappingSet***$[Name][channelName][storageName][domain]$. Imports an external *Extensional MappingSet* from a data channel. Parameter *Name* is the name of the new *Extensional MappingSet*. Parameter *channelName* is the name of the external data channel. Parameter *storageName* is the name, in the external data channel, of the *Extensional MappingSet* to import. Parameter *domain* is a comma separated list of *Dimensions*, i.e., the domain of the *Extensional MappingSet*. Notice that all *Dimensions* within the domain must be already imported before importing the *Extensional MappingSet*.

- ***Product***$[Name](d_1,\ldots,d_n)$. Generates a new *Extensional MappingSet*, without *Extensional Mappings*, whose domain is the Cartesian product $d_1 \times \ldots \times d_n$. Parameter *Name* is the name of the new *Extensional MappingSet*. Each operand d_i is a *Dimension* produced by some operator.

- ***Product***(MS,d). Generates a result *Extensional MappingSet* whose domain is the Cartesian product of d with the domain of *MS*. All *Extensional Mappings* of *MS* are kept in the resulting *Extensional MappingSet*.

- ***ProjectMappingSet***$[Name][s_1,\ldots,s_n](MS)$. Generates a new *Extensional MappingSet* whose domain is equal to the domain of *MS* and with one *Extensional Mapping* for each s_i. Operand *MS* is an *Extensional MappingSet* produced by some operator. Parameter *Name* is the name of the new *Extensional MappingSet*. Each parameter s_i is either a *Dimension* or an *Extensional Mapping* of *MS*.

- ***EvaluateIntensionalMappings***$[m_1,\ldots,m_n](MS)$. Adds new *Extensional Mappings* to *MS*. Operand *MS* is an *Extensional MappingSet* produced by some operator. Each m_i is an *intensional mapping expression* of the form $newMappingName = pm(s_1,\ldots,s_m)$, where *pm* is the name of a *primitive* mapping and each s_i is the name of either a *Dimension* or an *Extensional Mapping* of *MS*. The actual expression of m_i is generated from the expression of each s_i obtained from *MS*. If some *Extensional Mapping* of *MS* has been already computed by an expression equivalent to the actual expression of m_i, the new name *newMappingName* is added to the list of names referencing such *Extensional Mapping*. Otherwise, the *primitive* mapping is evaluated for each element of the domain of *MS* to produce a new *Extensional Mapping* called *newMappingName*.

- ***EvaluateExtensionalMapping***$[m](MS)$. Appends a new *Extensional Mapping* to *MS* (an *Extensional MappingSet* produced by some operator). Parameter *m* is an *extensional mapping expression* of the form $newMappingName = ems.em(s_1,\ldots,s_m)$, where *ems* references an *Extensional MappingSet*, *em* references an *Extensional Mapping* of *ems*, and each s_i is the name of either a *Dimension* or an *Extensional Mapping* of *MS*. The domain of *ems* must be defined by the Cartesian product of *m* dimensions $d_1 \times d_2 \times \ldots \times d_m$ in such a way that the data type of each s_i is compatible with the data type of each d_i. If some *Extensional Mapping* of *MS* has been computed by an expression equivalent to *m*, then the name *newMapping* will be added to the names of such *Extensional Mapping*, otherwise *em* will be evaluated.

- ***EvaluateConstant***$[Name](MS,k)$. Generates a new *Extensional Mapping* from the expression of k. Parameter *Name* is the name of the new *Extensional Mapping*. Operand *MS* is an *Extensional MappingSet* produced by some operator. Operand k is a *Constant* value. If the expression of k is already present in some *Extensional Mapping* of *MS*, then the name *newMapping* will be added to such *Extensional Mapping*. Otherwise, a new *Extensional Mapping* called *newMapping* is added to *MS*, which records the value obtained from k for each element of the domain. The data type and expression of the new mapping will be obtained from k. The domain of the new mapping in *MS* will be empty.

- ***EvaluateAggregateMappings***$[Name][groupBy][orderBy][c][ag_1,\ldots,ag_n](MS)$. *MS* is an *Extensional MappingSet* obtained from some operator. Parameter *groupBy* is a list of names of *Dimensions* of *MS*. Parameter *ordSpec* is an optional ordering specification composed of a list of pairs (s,o), where each s is the name of either a *Dimension* or an *Extensional Mapping* of *MS*, and o is an ordering direction, either *ascending* or *descending*. Optional parameter c is the name of an *Extensional Mapping* of *MS* of *Boolean* data type. Each ag_i is an expression of the form $newMapping_i = AggMapping_i(s_1,s_2,\ldots,s_n)$, where $AggMapping_i$ is the name of a *primitive* aggregate mapping and each s_j is the name of either a *Dimension* or an *Extensional MappingSet* of *MS*. The result *Extensional MappingSet* will have as domain the list of *Dimensions* referenced in *groupBy* and as *Extensional Mappings* the list of mappings of *MS* whose domain does not contain *Dimensions* not present in *groupBy*, together with the new mappings generated by $ag_1,ag_2,\ldots,ag_n$. To achieve this, first *MS* is grouped by *Dimensions* in *groupBy* and *Extensional Mappings* whose domain does not contain *Dimensions* out of *groupBy*. Next, the sequence of tuples of each group is filtered using c. Then, the result is ordered according to *ordSpec*. Finally, each aggregate mapping is evaluated in the ordered sequence of tuples to produce just one value for each group. If some *Extensional Mapping* of *MS* already has the same expression of one of the aggregates $aggMapping_i$ to be evaluated, then *newMapping* will be added to the names of such *Extensional Mapping* and $aggMapping_i$ will not be evaluated again.

- ***StoreMappingSet***$[Name](MS)$. Parameter *Name* is a *CString* and operand *MS* is an *Extensional MappingSet* obtained from some operation. The metadata of *MS* and the data of each of its *Extensional Mappings* is saved to disk. This operation has not effect if *MS* does not have any *Extensional Mapping*.

– ***ExportMappingSet***[*channelName*][*Name*](*MS*). Exports the *Extensional MappingSet* *MS* to an external data channel. Parameter *MS* is an *Extensional MappingSet* generated by some operator. Parameter *channelName* is the name of the external data channel. Parameter *Name* is the name, in the external data channel, of the new exported *Extensional MappingSet*.

Constant Operators

– ***Literal***[*Name*][*l*]. Transforms a literal into a data value that is recorded in the result *Constant*. Parameter *Name* is the name of the new generated *Constant*. Parameter *l* is the CString representation of a literal. The expression associated to the *Constant* will be *l*.

– ***ImportConstant***[*Name*][*channelName*][*storageName*]. Imports a *Constant* from an external data channel. Parameter *Name* is the name of the new *Constant*. Parameter *channelName* is the name of the external data channel. Parameter *storageName* is the name, in the external data channel, of the *Constant* to import.

– ***ScanConstant***[*Name*]. Reads a *Constant* from local catalog. Parameter *Name* is the name of the *Constant* to read.

– ***EvaluateIntensionalMapping***[*Name*][*m*]($k_1, \ldots, k_n$). Each operand k_i is a *Constant* obtained from some operation and *m* is the name of a *primitive* mapping. Mapping *m* is evaluated using the values of k_i as parameters to obtain the value for the result *Constant*. The expression of the result will also be generated from *m* and the expression of each k_i.

– ***EvaluateExtensionalMapping***[*Name*][*m*]($k_1, \ldots, k_n$). It is similar to the above operation, however now *m* references an *Extensional Mapping* of an *Extensional MappingSet*. The data of both *m* and the *Dimensions* of the domain of *m* have to be accessed to obtain the result value. The expression of the result will be generated from *m* and the expression of each k_i.

– ***EvaluateAggregateMapping***[*Name*][*orderBy*][*c*][*agg*](*MS*). *MS* is an *Extensional MappingSet* obtained from some operation. Parameter *ordSpec* is an ordering specification composed of either *Dimensions* or *Extensional Mappings* of *MS* and ordering directions (*ascending* or *descending*). Parameter *c* is an *Extensional Mapping* of *MS* of *Boolean*

type. Parameter *agg* is an expression of the form $aggMapping(s_1, s_2, \ldots, s_n)$, where *aggMapping* is a *primitive* aggregate mapping and each s_i references either a *Dimension* or an *Extensional Mapping* of *MS*. The elements of *MS* are first filtered using *c* and next ordered according to *ordSpec*. Finally, the aggregate mapping is applied to reduce *MS* to a single value of the result *Constant*. The expression of the result *Constant* will be generated from the expression of the aggregate mapping, by concatenating at the appropriate place the expression of each s_i obtained from *MS*.

– ***StoreConstant***$[Name](k)$. Saves *Constant k* to disk using *Name* as its storage name. Parameter *Name* is a *CString* and operand *k* is a *Constant* generated by some operator.

– ***ExportConstant***$[channelName][Name](k)$. Exports the *Constant k* to an external data channel. Parameter *k* is a *Constant* generated by some operator. Parameter *channelName* is the name of the external data channel. Parameter *Name* is the name, in the external data channel, of the new exported *Constant*.

4.3.4 Evaluation of MAPAL Expressions

To illustrate how MAPAL expressions are transformed to MAPAL operators, the sequence of operators corresponding to the evaluation of the following MAPAL expressions is given next.

```
<IntensionalMapping name="RiskByMunicipality" domain="m,t">
  <ForEach var="p"> Loc5m </ForEach>
  <Where> within(p, Municipality.Geo(m)) </Where>
  <Aggregate> AVG(ForestFire.Risk(p, t)) </Aggregate>
</IntensionalMapping>

<ExtensionalMappingSet name="MunicipalityWithRisk" storeName="MunicipalityWithRisk"
                       domain="m MunCode, t ObsDate">
  <ExtensionalMapping name="Name"> Municipality.Name(m) </ExtensionalMapping>
  <ExtensionalMapping name="Geo"> Municipality.Geo(m) </ExtensionalMapping>
  <ExtensionalMapping name="Risk"> RiskBymunicipality(m, t) </ExtensionalMapping>
</ExtensionalMappingSet>
```

Extensional MappingSet `MunicipalityWithRisk` provides the fire risk index for municipalities at every stored time instant. The name and geometry of each municipality is provided by accessing *Extensional MappingSet* `Municipality`. The fire risk index for each municipality is calculated by *Intensional Mapping* `RiskByMunicipality` as the average of the fire risk indexes for all points within the municipality geometry. The fire risk index

for a specific municipality at a specific time instant is provided by *Extensional MappingSet* `ForestFire` defined in Section 4.3.1.

The following operator expressions are generated to evaluate the above MAPAL expressions:

```
1  d1 = ScanDimension[MunCode]
2  d2 = ScanDimension[ObsDate]
3  MS1 = Product[MunicipalityWithRisk](d1,d2)
4  MS2 = EvaluateExtensionalMapping[Name=Municipality.Name(MunCode)](MS1)
5  MS3 = EvaluateExtensionalMapping[Geo=Municipality.Geo(MunCode)](MS2)
6  d3 = ScanDimension[Loc5m]
7  MS4 = Product(MS3, d3)
8  MS5 = EvaluateExtensionalMapping[M1=Municipality.Geo(MunCode)](MS4)
9  MS6 = EvaluateIntensionalMapping[M2=within(Loc5m,M1)](MS5)
10  MS7 = EvaluateExtensionalMapping[M3=ForestFire.Risk(Loc5m,ObsDate)](MS6)
11  MS8 = EvaluateAggregateMapping[MunCode,ObsDate][][M2][Risk=AVG(M3)](MS7)
12  MS9 = ProjectMappingSet[Name,Geo,Risk](MS8)
13  MS10 = StoreMappingSet[MunicipalityWithRisk](MS9)
```

First, *Dimensions* `MunCode` and `ObsDate` are read from local catalog (1-2). Notice that either imported from an external data channel or generated by previous operations, they must be already stored in local catalog. Then, *Extensional MappingSet* `MunicipalityWithRisk` is generated by building a new domain from `MunCode` and `ObsDate` (3). *Extensional Mappings* `Name` and `Geo` are added to `MunicipalityWithRisk` by evaluating *Extensional Mappings* `Municipality.Name` and `Municipality.Geo`, respectively (4-5). In order to generate *Extensional Mapping* `Risk`, *Intensional Mapping* `RiskByMunicipality` has to be evaluated. Thus, *Dimension* `Loc5m` is read from local catalog (6) and added to the domain of `MunicipalityWithRisk` (7). A temporary mapping `M2` storing the evaluation of the condition defined in element `<Where>` of `RiskByMunicipality` is required to calculate the aggregated function AVG. Such evaluation is performed by evaluating *Extensional Mapping* `Municipality.Geo` (8) and *Intensional Mapping* `within` (9). The fire risk index for each location and time instant is calculated by evaluating *Extensional Mapping* `ForestFire.Risk` and stored as a temporary mapping `M3` (10). *Extensional Mapping* `Risk` storing the aggregated average indexes is generated in (11) by evaluating an *Aggregate Mapping*. Notice that *Dimension* `Loc5m` has been deleted from the domain of the resulting *Extensional MappingSet*, temporary mapping `M2` is used as the aggregate condition and *Extensional Mapping* `Risk` is calculated by calling the *primitive* aggregate function AVG over temporary mapping `M3`.

Finally, only *Extensional Mappings* `Name`, `Geo` and `Risk` are considered in the resulting *Extensional MappingSet* `MunicipalityWithRisk` (12) and local catalog is updated (13).

CHAPTER 5

MAPAL IMPLEMENTATION

5.1 Introduction

Column-oriented DBMSs have shown to be the most efficient tools for the implementation of OLAP (On-Line Analytical Processing) over very large data warehouses. It is noticed that OLAP performs data reading tasks in workloads where only some columns of the tables are involved. In that case, recording data columnwise avoids having to read the whole table to access only some columns and furthermore, it enables better leveraging of compression techniques [3].

A column-oriented prototype has been developed to implement a MAPAL system. A general overview of the architecture of such prototype is depicted in Fig. 5.1. *Observation Data Analysis Service* receives a set of *Operator Expressions*[1] as input. Such *Operator Expressions* determine the *Operator Tree* to be processed by the *Query Processor*. A SODA implementation should compile MAPAL expressions provided by users, build several query execution plans in cooperation with a *Query Optimizer*, select the most appropriate alternative and execute the corresponding *Operator Expressions*. Taking into account that the major objective of this prototype is to show the capabilities of SODA regarding observation data analysis performance, the implementation of a MAPAL compiler, a *Query Optimizer* and a complex *Query Processor* goes far beyond the scope of this prototype. Likewise, implementation of compilers for XODDL and *internal* process definitions are out of the scope of this prototype. Thus, administration staff must manually configure the *Catalog Manager*, which provides

[1] As defined in Section 4.3.3

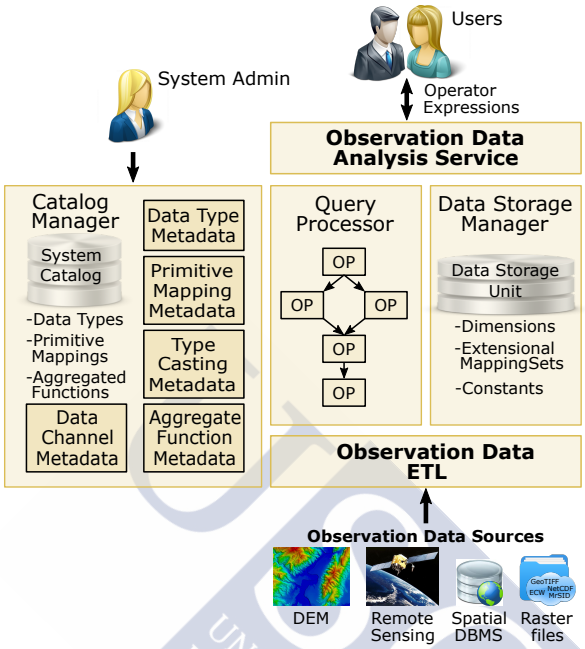


Figure 5.1: SODA Prototype Architecture.

access to all the system metadata, and records system provided constructors, including data types, *primitive mappings* and operators, *aggregate functions* and type castings. *Dimensions*, *Extensional MappingSets* and *Constants* are recorded in a *Data Storage Unit*. Finally, the *Observation Data ETL* module enables the importing of *external* observation data.

Apache Parquet [1] and Apache Spark [10] are two very well known tools of the Apache Hadoop ecosystem. Apache Parquet is an efficient columnar data storage technology that enables the recording of complex nested data structures. It supports efficient compression and encoding schemes and uses the record shredding and assembly algorithm described in [74].

Apache Spark is a fast and general-purpose cluster computing system, providing APIs for various programming languages and an optimized engine. Its primitive data structure is the Resilient Distributed Dataset (RDD), which can be created from many storage formats. An RDD is a collection of data elements partitioned across the nodes of the cluster that can be operated on in parallel. Many primitive operations are supported on RDDs, including input/output operations, application of functions to RDD elements (*Map* operation) and var-

ious relational operations (e.g., *select*, *join*, *union*, *intersection*). Operation *Range* enables the generation of series of integer numbers in a result RDD. Operation *ZipWithUniqueId* enables the generation of unique identifiers for RDD elements at each computing node. A Spark *DataFrame* is constructed by combining an RDD with a schema that defines attribute names and types. SQL operations are supported on Spark *DataFrames*, e.g., *select*, *where*, *join*, *group by*, *aggregation*, *union*, *intersection*. *DataFrames* can be straightforwardly created/stored from/to Apache Parquet files.

The above characteristics make the combination of Spark with Parquet a very good candidate platform for the efficient implementation of the proposed column oriented MAPAL prototype. Based on my background on computer programming, the Spark Java API has been selected to develop the prototype software code. And thus, both data types and data structures have also been coded in Java.

The following sections provide an overview of the design of both physical data structures and operations for the efficient implementation of MAPAL on top of the aforementioned column-oriented storage and processing framework. The remainder of this Chapter is organized as follows. Section 5.2 exposes the implementation of data types defined in Section 4.2.1. Implementation of data structures also defined in Section 4.2.1 is explained in Section 5.3. The implementation of required Spark user defined types and functions is covered in Section 5.4. Section 5.5 is devoted to MAPAL ETL structures that enable efficient data import and export. Implementation of operators defined in Section 4.3.3 is covered in Section 5.6. Finally, MAPAL performance is evaluated in Section 5.7.

5.2 Data Types Implementation

An original contribution of this Thesis is the definition of spatial and temporal data types that enable the integrated and uniform representation of spatio-temporal entities and spatio-temporal coverages (time evolving *raster* data). This section is devoted to the implementation of all the data types defined in Section 4.2.1. Each *primitive* mapping supplied by the system is implemented by a relevant data type class.

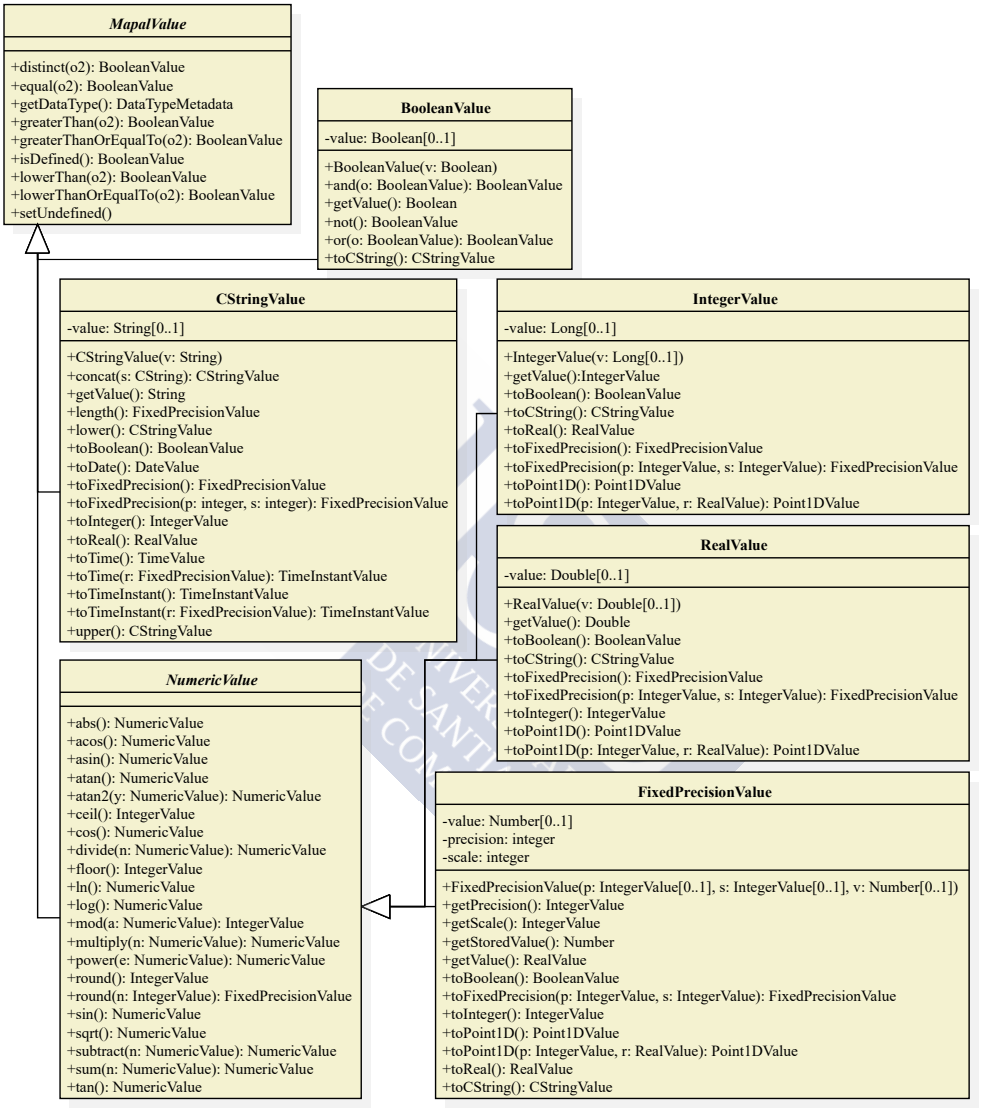


Figure 5.2: Class diagram of Conventional data types in the prototype implementation.

5.2.1 Conventional Data Types Implementation

The class diagram of conventional MAPAL data types implemented in the developed prototype is depicted in Fig. 5.2. An abstract class *MapalValue* encapsulates the common operations (*primitive mappings*) that every data type must implement, as defined in Table 4.1. Thus, the rest of the classes representing MAPAL data types must inherit from *MapalValue*. Class *BooleanValue* has an attribute *value* of Java type *Boolean* to store the actual boolean value. Moreover, *BooleanValue* implements the primitive boolean mappings defined in Table A.1, and provides the required constructor and accessor methods. Similarly to *BooleanValue*, class *CStringValue* provides constructor and accessor methods, and implements the string primitive mappings defined in Table A.2. The Java type *String* is used to store the actual *CString* value.

Primitive mappings common for all numeric (*Integer*, *Real*, *FixedPrecision*) data types, defined in Table A.3, are encapsulated by the abstract class *NumericValue* from which all the numeric classes inherit. Attribute *value* of *IntegerValue* and *RealValue* are of *Long* and *Double* Java types, respectively. Whereas, to store the actual integer value of attribute *value* in *FixedPrecisionValue*, the abstract Java type *Number* is used. Depending on the combination of attributes *precision* and *scale*, the most appropriate Java integer type inheriting from *Number* (*Byte*, *Short*, *Integer* and *Long*) is actually used. Furthermore, constructors, accessor methods and appropriate casting mappings are defined for all numeric data types.

Notice that constructor and accessor methods are defined for internal use within the implementation and are not accessible to MAPAL users through *primitive* mappings. Conventional data type values are generated automatically by the system from literal values in MAPAL sentences.

5.2.2 Temporal Data Types Implementation

A major added value in MAPAL are those data types that enable the definition of temporal *Samplings*. Abstract class *SamplingValue* in Fig. 5.3 defines the primitive mapping *subtract* for all those data types that may be used in *Sampling* definitions.

Inheriting from *SamplingValue*, abstract class *TemporalValue* encapsulates attributes and mappings common for all defined temporal data types. Specifically, attribute *resolution* uses the Java primitive type *double* to store the temporal resolution. Moreover, the mapping *sum* and an overloaded version of mapping *subtract*, defined in Table A.4, are added here together with the accessor method for attribute *resolution*. Similarly to numeric data types, *TimeValue*

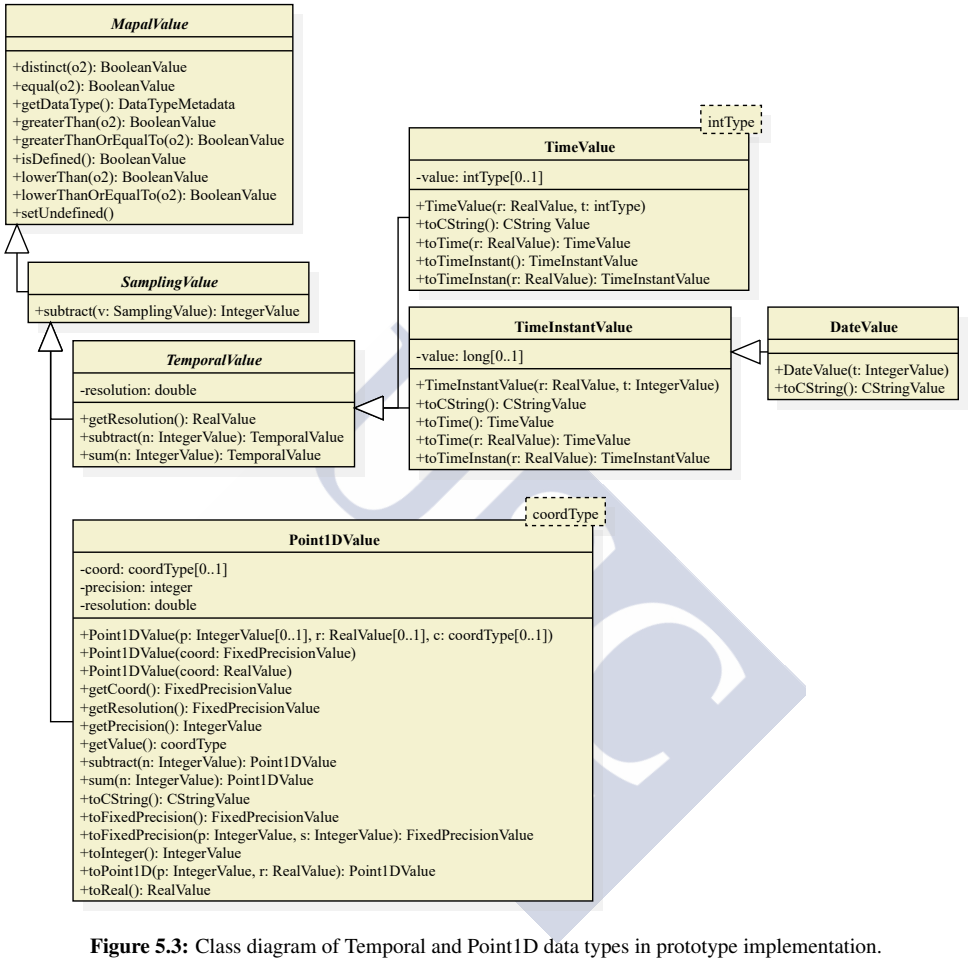


Figure 5.3: Class diagram of Temporal and Point1D data types in prototype implementation.

and *TimeInstantValue* implement constructors and the appropriate casting mappings also defined in Table A.4. While Java primitive type *long* is used to store the integer temporal value in *TimeInstantValue*, class *TimeValue* uses the most appropriate Java primitive integer type (*intType*) to store such integer value. The selection of *intType* is strongly dependent on the attribute *resolution*.

Since MAPAL data type *Date* is a shortcut for *TimeInstant(86400)*, class *DateValue* only defines a constructor and an overloaded version of mapping *toCString*. Similarly to conventional data types, constructor and accessor methods are only for internal use and are not

accessible by MAPAL users. The system generates them from literal values in MAPAL sentences.

5.2.3 *Point1D* Data Type Implementation

Another important MAPAL data type contributed by this Thesis is *Point1D*. Definition of spatial 1D *Samplings* is enabled by this data type. Thus, class *Point1DValue* in Fig. 5.3, also inheriting from *SamplingValue*, stores attributes *precision* and *resolution* using Java primitive types *integer* and *double*, respectively. As *TimeValue* does, the most appropriate Java primitive integer type (*coordType*) is used to store the integer spatial attribute *coord*. In this case, such Java type is selected depending on the value of attributes *precision* and *resolution*. Primitive *Point1D* mappings defined in Table A.5 are also present in *Point1DValue* together with constructors and accessor methods. As previous data types, such constructor and accessor methods are not accessible by MAPAL users.

5.2.4 *Point2D* Data Type Implementation

Probably the most important contribution of MAPAL, regarding data types, is the definition of data type *Point2D* that enables, in turn, the definition of 2D spatial *Samplings*. Class *Point2DValue* in Fig. 5.4 encapsulates the *Point2D* mappings defined in Table A.6, together with constructors and accessor methods.

Attributes *precision* and *resolution* store relevant *Point2D* parameters using Java primitive types *integer* and *double*, respectively. Attribute *value* uses a JTS[2] [60] object *Point* to store the 2D spatial coordinates. Since MAPAL data type *Point2D* also represents a geometry, *Point2DValue* implements the primitive mappings defined in *Geometry2DInterface*, which contains the vast majority of primitive mappings defined for all *Geometries* in Table A.7. Recall that *Point2D* enables the definition of 2D spatial *Samplings*, thus *Point2DValue* inherits from *SamplingValue* as well. *Point2DValue* constructor and accessor methods are also not accessible by MAPAL users.

5.2.5 *Geometric* Data Type Implementation

The developed prototype also implements the geometric data types defined in Section 4.2.1. Thus, class *Geometry2DValue* in Fig. 5.4 provides constructors and accessor methods to create

[2] Java Topology Suite.

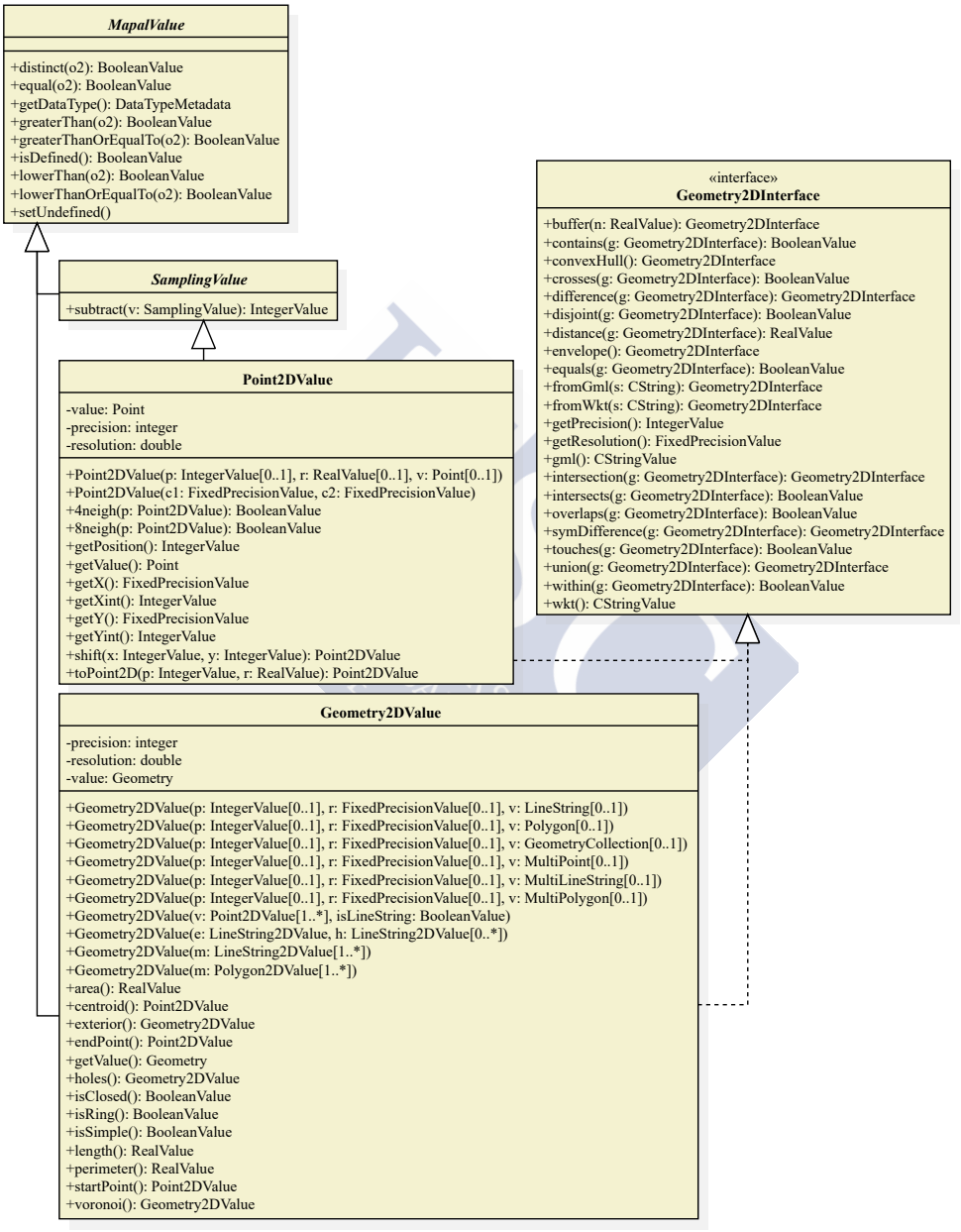


Figure 5.4: Class diagram of Point2D and Geometric data types in prototype implementation.

and access geometries, respectively. Furthermore, primitive mappings defined in Table A.8, Table A.9, Table A.10 and Table A.11 are also incorporated into *Geometry2DValue*. Property *value* uses a JTS abstract object *Geometry* to store the geometry value. Current implemented[3] subclasses of *Geometry* include *LineString*, *Polygon*, *MultiLineString*, *MultiPoint* and *MultiPolygon*. Even though all mentioned primitive mappings are implemented in *Geometry2DValue*, only appropriate ones may be represented depending on the actual subclass of *Geometry*. Similarly to *Point2DValue*, *Geometry2DValue* stores geometry *precision* and *resolution*, and implements the primitive mappings defined in *Geometry2DInterface*.

5.3 Data Structures Implementation

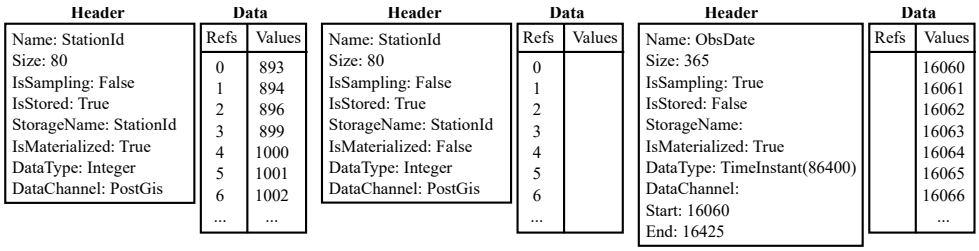
Efficient structures are required for recording data and metadata related to *Dimensions*, *Extensional MappingSets* and *Constants* both in disk and main memory. A detailed description of such structures implemented in the developed prototype is provided in this section.

5.3.1 In-Memory Structures Implementation

Each *Dimension*, *Extensional MappingSet* and *Constant*, either obtained from disk or calculated, is recorded in main memory in a structure composed of a header, with appropriate metadata, and a data area.

Metadata recorded in a *Dimension* header include name, size and data type. A *Dimension* might be obtained from disk or generated in memory as a result of some operation. A boolean *IsStored* is kept in the header to identify these two types of *Dimensions*. Attribute *StorageName* is used to identify *Dimensions* stored in the local catalog (introduced in next Section). Boolean attribute *IsMaterialized* shows whether a *Dimension* is materialized in main memory or not. A non-materialized *Dimension* (Fig. 5.5(b)) only records unique element identifiers in main memory, which reference element positions. Such references are generated in main memory using the size of the *Dimension*. A materialized *Dimension* might have been generated in memory, in such case it has only data values (Fig. 5.5(c)), or it may have been read from disk, in such case it has both references and data values (Fig. 5.5(a)). Non-materialized stored *Dimensions* enable the implementation of late materialization [2], which avoids having to read data values from disk which are not involved in any calculation.

[3] http://locationtech.github.io/jts/javadoc/.



(a) Materialized stored Dimension (b) Non-materialized stored Dimension (c) Materialized non-stored Dimension

Figure 5.5: Example of in-memory Dimension structures.

An *Extensional MappingSet* header must record global metadata (*Name*, *IsStored* and *StorageName*) as well as metadata of building *Dimensions* and *Extensional Mappings*. Fig. 5.6 illustrates an *Extensional MappingSet* computed in memory to record the temperature, elevation and location of each meteorological station at each observation date. *Extensional Mapping Temperature* records the result of the evaluation of the expression: $Observation.Temperature(StationId, ObsDate)$. *Extensional Mapping m1* is a temporary one that records the result of the expression $Station.Loc(StationId)$. *Extensional Mapping Loc* shares both data and metadata with *m1*, therefore it will share also the same header entry. *Extensional Mapping m2* is also temporary and records the result of the expression: $cast(m1(StationId, ObsDate)\ AS\ Point2D(7,5))$. Finally, *Extensional Mapping Elevation* records the result of the expression $Topo.Elevation(m2(StationId, ObsDate))$. It is noticed that *Extensional Mapping m2* records *Point2D*(7,5) values that reference elements in *Dimension Loc5m*. These references are needed to obtain the elevation values from disk. It is therefore noticed that beyond the references and values of the *Dimensions* and the values of each *Extensional Mapping*, both *Dimensions* and *Extensional Mappings* might record additional columns that contain references to stored *Dimensions*. Besides, the header of each *Extensional Mapping* keeps record of the subset of *Dimensions* from which it is dependent, i.e., its real domain and the expression that was used to compute it. The former is used during aggregate operations, as it will be shown in Section 5.6, whereas the latter helps in avoiding the computation of duplicate *Extensional Mappings* in the same *Extensional MappingSet* as it is the case of *Extensional Mappings m1* and *Loc*.

Header

Name: TemperatureElevationAtStation IsStored: False StorageName:

Dimensions

Name	IsSampling	IsStored	StorageName	IsMaterialized	DataType	Start	End	Size	DataChannel
StationId	False	True	StationId	False	Integer			80	PostGIS
ObsDate	True	True	ObsDate	False	TimeInstant(86400)	16060	16424	365	PostGIS

Mappings

Name	DataType	Domain	Expression
{Temperature}	FixedPrecision(5,2)	{StationId, ObsDate}	Observation.Temperature(StationId, ObsDate)
{m1, Loc}	Point2D(9, 0.01)	{StationId}	Station.Loc(StationId)
{m2}	Point2D(7,5)	{StationId}	Cast(Station.Loc(StationId), "Point2D(7,5)")
{Elevation}	FixedPrecision(7,3)	{StationId}	Topo.Elevation(m2)

RefDims

ReferencedBy	RefDimensions
m2	{Loc5m}

Data

StationId		ObsDate		Temperature	m1	m2		Elevation
Refs	Values	Refs	Values	Values	Values	Values	Loc5m	Values
0		0		1124	(51320501, 479952838)	(102641, 959906)	1388891280	40271
0		1		1136	(51320501, 479952838)	(102641, 959906)	1388891280	40271
0		2		1206	(51320501, 479952838)	(102641, 959906)	1388891280	40271
...		...		...	...	...	...	...
1		0		1645	(53881597, 475086604)	(107763, 950173)	1001420550	332839
1		1		1705	(53881597, 475086604)	(107763, 950173)	1001420550	332839
...		...		...	...	...	...	...

Figure 5.6: Example of in-memory MappingSet structures.

The structure of each *Constant* will record the following attributes: name, storage name, *IsStored* (whether the *Constant* has been stored), data value and the expression evaluated to generate the data value (used to avoid duplicate computations).

These in-memory structures for the representation of *Dimensions*, *Extensional MappingSets* and *Constants* during the execution of operations are implemented in Java, making use of the DataFrame structure of Spark to record data columns. Fig. 5.7 depicts an UML diagram of the *Dimension*, *Extensional MappingSet* and *Constant* structures designed for the implementation.

Each *Constant* resulting from some MAPAL operation is represented with an object of class *Constant*, Fig. 5.7. Notice that, together with relevant names, this class has also attributes to represent both its data value (of class *MapalValue*) and the expression used to compute it. As explained in Section 5.2.1, class *MapalValue* is the root of a hierarchy of classes that enable the representation of all the system data types. Each subclass of the hierarchy provides an appropriate data structure for the data value and a collection of methods to implement required

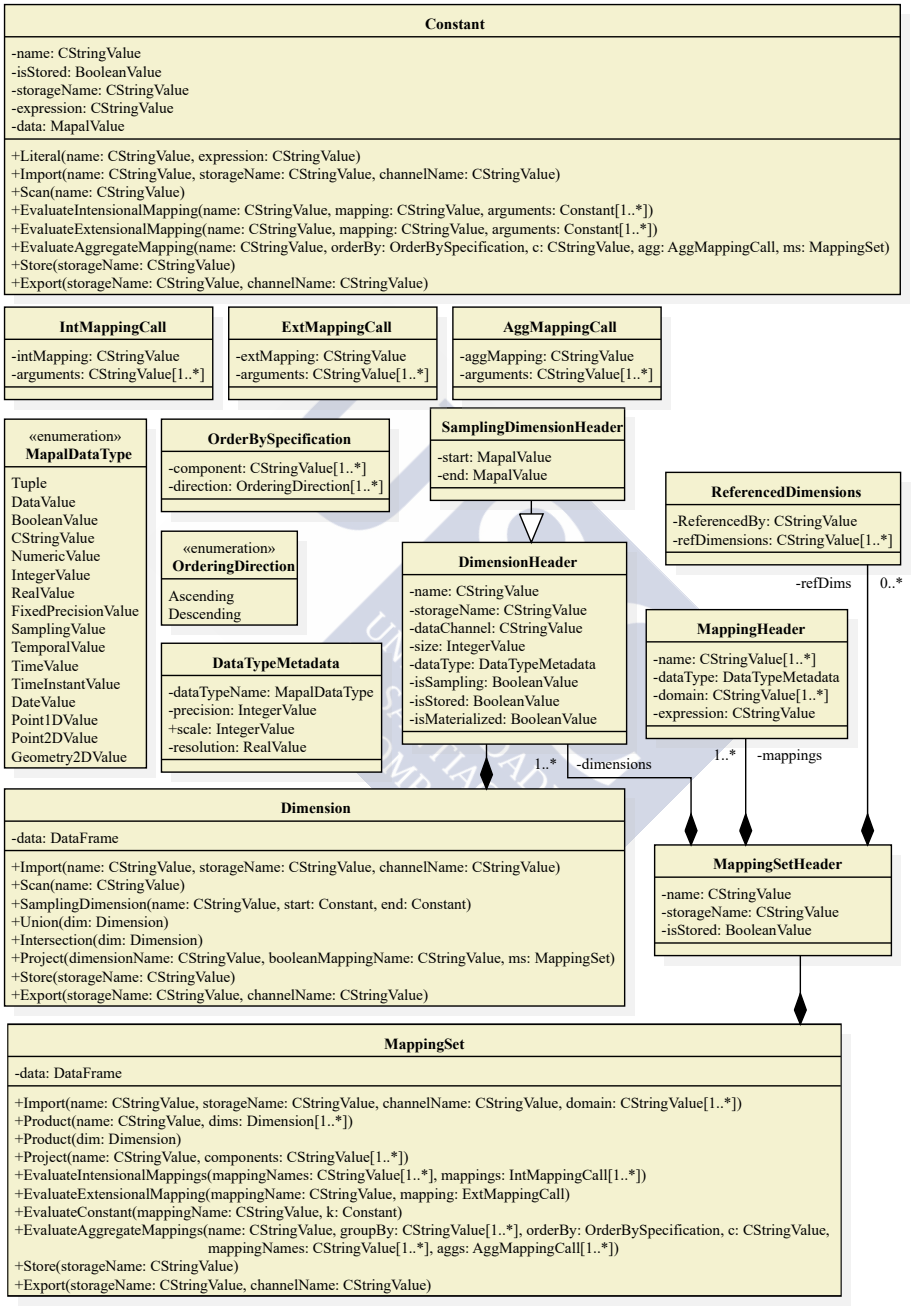


Figure 5.7: In-memory structures implementation with Spark.

primitive mappings and operators. Each *Constant* operation described in Subsection 4.3.3 has a relevant method in class *Constant*.

Each *Dimension* generated by some operation is represented in memory by an instance of class *Dimension*, Fig. 5.7. The header of class *Dimension* is an instance of class *DimensionHeader*. Notice that class *SamplingDimensionHeader* is provided for the proper representation of *SamplingDimensions*. *Dimension* data is stored in a Spark DataFrame, which may have one or two columns depending on whether the *Dimension* is stored or/and materialized, as shown in the example of Fig. 5.5. Each *Dimension* operator described in Subsection 4.3.3 is implemented with a relevant method in class *Dimension*.

Each *Extensional MappingSet* generated by some operation is stored in memory by an instance of class *MappingSet*. The header is an instance of type *MappingSetHeader*, which includes one *DimensionHeader* per *Dimension* of the *Extensional MappingSet* and one *MappingHeader* per *Extensional Mapping*. Information related to which *Extensional Mapping* or *Dimension* references values of stored *Dimensions* is provided by instances of class *ReferencedDimensions*. All the data columns of an *Extensional MappingSet* are represented in a single DataFrame. Such DataFrame includes columns for *Dimension* references and *Dimension* values, for *Extensional Mapping* values and for *Dimensions* referenced by either *Extensional Mappings* or *Dimensions*. Each *Extensional MappingSet* operation described in Subsection 4.3.3 is implemented by a relevant method in class *MappingSet*.

Additional classes have been defined to ease the proper representation of several class attributes and method arguments. Thus, attribute *dataType* in *DimensionHeader* and *MappingHeader* stores information of recorded data type in a *DataTypeMetadata* object, which in turn stores the data type name in a *MapalDataType* object. Optional argument *orderBy*, in methods for calculating aggregate mappings both in *Constant* and *MappingSet* classes, stores the ordering specification in an *OrderBySpecification* object, which in turn stores the ordering direction in an *OrderingDirection* object. Classes *IntMappingCall*, *ExtMappingCall* and *AggMappingCall* enable the representation of expressions of the form $mapping(a_1, \cdots, a_n)$ used in the evaluation of *intensional*, *extensional* and *aggregate* mappings, respectively.

Spark class DataFrame provides methods to properly depict the DataFrame schema (`printSchema()`) as well as the recorded DataFrame data (`show()`). For illustration purposes, Fig. 5.8 depicts the output of such methods in the stored DataFrame of *Extensional MappingSet Municipality*. Notice that *Dimensions* and *Extensional Mappings* values are recorded

```
+------------+--------------+-----------+-----------------+-----------------+
|MunCode_refs|MunCode_values|Name_values|Population_values|       Geo_values|
+------------+--------------+-----------+-----------------+-----------------+
|           3|         15004|       Ares|             5801|Geometry2DValu...|
|         117|         15008|   Bergondo|             6738|Geometry2DValu...|
|           5|         15011|      Boiro|            19144|Geometry2DValu...|
|           8|         15015|    Cabanas|             3287|Geometry2DValu...|
|          13|         15019|   Carballo|            31366|Geometry2DValu...|
|          16|         15022|    Cedeira|             7246|Geometry2DValu...|
|         133|         15033|      Dodro|             2934|Geometry2DValu...|
|         135|         15037|   Fisterra|             4907|Geometry2DValu...|
|         138|         15040|       Laxe|             3267|Geometry2DValu...|
|          35|         15044|      Mañón|             1507|Geometry2DValu...|
|          38|         15048|       Miño|             5786|Geometry2DValu...|
|          40|         15051|   Mugardos|             5470|Geometry2DValu...|
|          44|         15055|       Neda|             5413|Geometry2DValu...|
|          48|         15059|      Ordes|            12939|Geometry2DValu...|
|          51|         15062|      Outes|             7010|Geometry2DValu...|
|          54|         15066|     O Pino|             4708|Geometry2DValu...|
|          61|         15073|    Ribeira|            27811|Geometry2DValu...|
|          63|         15077|Santa Comba|             9913|Geometry2DValu...|
|          66|         15080|    Sobrado|             2016|Geometry2DValu...|
|          10|         15084|    Tordoia|             3817|Geometry2DValu...|
+------------+--------------+-----------+-----------------+-----------------+
only showing top 20 rows

root
 |-- MunCode_refs: long (nullable = false)
 |-- MunCode_values: string (nullable = false)
 |-- Name_values: string (nullable = false)
 |-- Population_values: long (nullable = false)
 |-- Geo_values: geometry2dvalue (nullable = false)
```

Figure 5.8: In-memory DataFrame schema and partial data of Extensional MappingSet *Municipality*.

in DataFrames using *User-Defined Types* (UDTs). Section 5.4.1 provides a detailed description of such special data types.

5.3.2 Disk Structures Implementation

A local catalog with relevant metadata of *Dimensions*, *Extensional MappingsSets* and *Constants* is recorded in disk and loaded in main memory on system startup. The schema of the catalog, filled with example metadata, is shown in Fig. 5.9.

Stored data for each *Dimension* include name, data type metadata and size (number of elements). A boolean property specifies whether the *Dimension* is a *sampling* or not. For *sampling Dimensions*, the start and end values of the *sampling* are also recorded in the catalog. Notice that only temporal and spatial *samplings* are allowed and, in any case, always integer coordinates in the underlying grid are recorded. For non sampling *Dimensions*, the path to the file where the *Dimension* data is stored is also recorded. For each *Extensional MappingSet* the catalog records its name, the set of *Dimensions* that define the domain of its mappings,

Dimensions

Name	DataType	Size	Sampling	Start	End	FilePath
StationId	Integer	80	false			/pathToFile/...
ObsDate	TimeInstant(86400)	365	true	16060	16424	
Loc5m	Point2D(7,5)	1845147150	true	218500339171158	219427339164617	
MunCode	CString	314	false			/pathToFile/...

MappingSets

Name	Dimensions	FilePath
Station	{StationId}	/pathToFile/...
Observation	{StationId, ObsDate}	/pathToFile/...
Topo	{Loc5m}	/pathToFile/...
Municipality	{MunCode}	/pathToFile/...

Mappings

MappingSet	Mapping	DataType
Station	Name	CString
Station	Loc	Point2D(9, 0.01)
Observation	Temperature	FixedPrecision(5, 2)
Observation	Humidity	FixedPrecision(2, 0)
Observation	WindSpeed	FixedPrecision(6, 3)
Topo	Elevation	FixedPrecision(7, 3)
Municipality	Name	CString
Municipality	Geo	MultiPolygon(9, 0.01)

Constants

Name	DataType	Value
IDWDistance	FixedPrecision(6, 0)	40000
MinTemperature	FixedPrecision(5, 2)	-1000
MaxTemperature	FixedPrecision(5, 2)	5000
TemperatureWeight	FixedPrecision(2, 2)	20
HumidityWeight	FixedPrecision(2, 2)	20
MinWindSpeed	FixedPrecision(6, 3)	0
MaxWindSpeed	FixedPrecision(6, 3)	60
WindSpeedWeight	FixedPrecision(2, 2)	30
MaxSlope	FixedPrecision(3, 3)	500
SlopeWeight	FixedPrecision(2, 2)	30

Figure 5.9: System Catalog example.

and also the path to the storage file. For each *Extensional Mapping* the catalog records its name, the name of the holding *Extensional MappingSet* and data type metadata. Both data and metadata (name and data type) of *Constants* are also recorded in the catalog.

Beyond the above metadata, the data of non sampling *Dimensions* and *Extensional MappingSets* must also be recorded in disk structures. Following the columnwise data storage paradigm, column oriented Parquet files are used to record each non sampling *Dimension* and each *Extensional MappingSet*. A two column file is generated to store each *Dimension*, recording data values and relevant references. First, a data column of increasing ordered values is generated from *Dimension* values. Then, a second column is generated by assigning a sequential reference to each value in turn. Storing references in disk ensures that the ordering process is executed only once in the storage step and not multiple times in the materializing step. Notice that one logical file storing a *Dimension* may be split into several physical files when is stored in the Hadoop distributed file system, and thus an ordered reading of *Dimension* values is not assured by HDFS readers. Contrary to classical relational approaches, that record *Tuples*, non sampling *Dimensions* are recorded here only once, independently of the number of *Extensional MappingSets* referencing them. Similarly to *Dimensions*, each *Extensional MappingSet* is stored in a Parquet file. In this case, the storage file has one data column

```
corresponding Parquet message type:
message spark_schema {
  optional int64 Refs;
  optional binary Name (UTF8);
  optional int64 Population;
  optional group Geo {
    required binary value;
    required int32 size;
    required double resolution;
  }
}
```

Figure 5.10: Parquet file schema of Extensional MappingSet *Municipality*.

for each *Extensional Mapping* within the *MappingSet*, and one additional column with references. First, an increasing ordering is applied to the *Extensional MappingSet* domain and a sequential reference is assigned to each ordered value in turn. Recall that each value in the *Extensional MappingSet* domain is assigned to a specific combination of *Extensional Mapping* values. Thus, each domain value is replaced by the relevant reference. Finally, one column per each *ExtensionalMapping* and an additional column with references are stored in the Parquet file. Storage of *Extensional MappingSet* references enables the following features:

– Only store the values of the Cartesian product of *Dimensions* that make sense for a specific *Extensional MappingSet* without using additional structures.

– Provide a mechanism to ensure both the ordering process is executed only once (in storage step) and the ordered materialization of *Extensional MappingSets* (in the reading step).

Notice that compression techniques and appropriate encoding systems enabled by the Parquet storage format will drastically reduce the payload of stored columns. To improve this payload reduction even more, a novel well known binary representation has been developed to store geometries (including *Point2DValues*) in DataFrames and consequently in Parquet files. Such representation improves the well known binary representation of OGC[4] geometries, enabling the encoding of building point coordinates as integer values rather than double precision real values.

For illustration purposes, Fig. 5.10 depicts the schema of the Parquet file generated to store *Extensional MappingSet Municipality* into the local catalog. Column *Geo* stores the required attributes to properly represent the geometry of each municipality. Notice that attribute *value*

[4] Open Geospatial Consortium.

in column *Geo* is stored using the Parquet type *binary* that enables the proper storage of the well known binary representation generated for each geometry.

5.4 User-Defined Data Types and Functions

5.4.1 User-Defined Data Types (UDTs)

Since *Dimension* and *Extensional Mapping* data values are stored in DataFrames, each *MapalValue* needs to be mapped to a relevant SparkSQL [14] data type, i.e., those data types defined in Spark to store data within a DataFrame. For those *user* data types too complex to be mapped to simple SparkSQL data types, appropriate SparkSQL *User-Defined Types* (UDTs) are defined. Therefore, relevant UDTs have been defined for every *MapalValue*. *Point2DValueUDT*, defining the UDT corresponding to *Point2DValue*, is shown in Code 5.1 for illustration purposes. Notice that *Point2DValueUDT* inherits from *MapalValueUDT* which in turn inherits from interface *UserDefinedType<MapalValue>*. Thus, the following four methods have to be implemented:

- `sqlType()`. Defines the underlying storage SparkSQL type for the UDT. *Point2DValueUDT* defines a *StructType* composed of three *StructFields*. Field *precision* enables the storage of the *Point2DValue* precision as an integer value. Field *resolution* enables the storage of the *Point2DValue* resolution as float-point double value. Field *value* enables the storage of the *Point2DValue* value as a binary value, i.e., an array of bytes. Recall that the *Point2DValue* value is an object of type *Point* as defined in JTS.

- `userClass()`. Returns the actual class object of the value to be stored. *Point2DValueUDT* obviously returns the class *Point2DValue*.

- `Serialize(Object userValue)`. Converts the user type to a SparkSQL type. *Point2DValueUDT* receives a Point2DValue as input and returns a *GenericInternalRow* composed of three values. An integer value storing the precision. A double value storing the resolution. A byte array storing the aforementioned well known binary representation of the JTS *Point*. Such byte array can be directly matched to a Parquet *binary* when stored to disk. Notice that the row structure must match the user data type defined in `sqlType()`.

```
/**
 * User-Defined Type to store Point2DValue data into a DataFrame column
 */
public class Point2DValueUDT extends MapalValueUDT {
  @Override
  public DataType sqlType() {
    List<StructField> fields = new ArrayList<>();
    fields.add(DataTypes.createStructField("value",
                                           DataTypes.BinaryType,
                                           false));
    fields.add(DataTypes.createStructField("precision",
                                           DataTypes.IntegerType,
                                           false));
    fields.add(DataTypes.createStructField("resolution",
                                           DataTypes.DoubleType,
                                           false));
    StructType structType = DataTypes.createStructType(fields);
    return structType;
  }

  @Override
  public Class<Point2DValue> userClass() {
    return Point2DValue.class;
  }

  @Override
  public Object serialize(Object value) {
    if (value instanceof Point2DValue){
      Point2DValue point2Dvalue = (Point2DValue)value;
      byte[] value = point2Dvalue.toWKB();
      Integer size = point2Dvalue.getPrecision();
      Double resolution = point2Dvalue.getResolution();
      Object[] values = {value, size, resolution};
      GenericInternalRow result = new GenericInternalRow(values);
      return result;
    }
    return null;
  }

  @Override
  public MapalValue deserialize(Object row) {
    if (row instanceof UnsafeRow){
      UnsafeRow unsafeRow = (UnsafeRow) row;
      try {
        return new Point2DValue(unsafeRow.getInt(1),
                                unsafeRow.getDouble(2)
                               ).fromWKB(unsafeRow.getBinary(0));
      } catch (ParseException e) {
          e.printStackTrace();
```

```
      }
    }
    else if (row instanceof GenericInternalRow){
      GenericInternalRow genericRow = (GenericInternalRow) row;
      try {
        return new Point2DValue(genericRow.getInt(1),
                                genericRow.getDouble(2)
                               ).fromWKB(genericRow.getBinary(0));
      } catch (ParseException e) {
          e.printStackTrace();
      }
    }
    return null;
  }
}
```

Code 5.1: User-Defined Type *Point2DValueUDT*.

```
/**
 *  UDF for primitive mapping greaterThan
 */
public class GreaterThan implements UDF2<MapalValue, MapalValue, BooleanValue>{
  @Override
  public BooleanValue call(MapalValue a, MapalValue b) throws Exception{
    return a.greaterThan(b);
  }
}
```

Code 5.2: User-defined Function for primitive mapping *greaterThan*.

– `Deserialize(Object userValue)`. Converts a SparkSQL type to a user type. Both *UnsafeRow* and *GenericInternalRow* can be received as input values in *Point2DValue-UDT*. Values within these rows are used to build the returned *Point2DValue*.

5.4.2 User-Defined Functions (UDFs)

Spark implements a wide range of system defined functions to be applied over data recorded in DataFrames that eases the processing of both primitive and user-defined data types. For those processing needs that go beyond the basic implemented functions, Spark provides a mechanism that enable users to define new functions, called *User-Defined Functions* (UDFs). Therefore, MAPAL leverages UDFs to implement its *primitive* mappings. Code 5.2 shows the UDF for the *primitive* mapping *greaterThan*. Notice that method *greaterThan* of input argument *a* is called to return the resulting *BooleanValue*.

```
/**
 *  Interface to import/export observation data from/to data channels
 */
public interface DataChannelInterface {
   public Dimension ImportDimension(String dimensionName, String storageName);
   public Constant ImportConstant(String constantName, String storageName);
   public MappingSet ImportMappingSet(String mappingSetName,
                                      String storageName,
                                      ArrayList<String> domain)
   public void ExportDimension(String dimensionName, Dimension dimension);
   public void ExportConstant(String constantName, Constant constant);
   public void ExportMappingSet(String storageName, MappingSet mappingSet);
}
```

Code 5.3: Interface *DataChannelInterface*.

5.5 Data Channels Implementation

As shown in Fig. 5.1, one of the main components of the MAPAL prototype architecture is the module *Observation Data ETL*. Such component must enable the importing and exporting of *heterogeneous* observation data. Therefore, appropriate pieces of code, called *data channels* in the context of this Thesis, can be added to the *Observation Data ETL* component in order to access external observation data sources. No restriction on communication between *data channels* and data sources is imposed in this prototype. Restrictions have to do with both implementation of *data channel* catalogs and communication between *data channels* and MAPAL core. Regarding the former, access to the storage file system from *data channel* code must observe the specific rules provided by the administration staff.

To enable a proper input/output communication between *heterogeneous data channels* and MAPAL core, the well known design pattern *Adapter* is adopted here to define the interface *DataChannelInterface* (Code 5.3) that must be implemented by every *data channel*. *DataChannelInterface* defines the required methods to import and export *Dimensions*, *Extensional MappingSets* and *Constants*. Notice that based on parameter *channelName* of relevant MAPAL operations, the required import/export method is called on the appropriate *data channel*. Thus, such parameter is not needed in *DataChannelInterface* methods.

Two *data channels* have been implemented in the proposed prototype. Channel *GeoTIFFChannel* enables the importing/exporting of observation data from/to GeoTIFF files [42]. GeoTIFF is a file format for saving *raster* images with embedded georeferencing information.

Such georeferencing information is used to assign each pixel in the GeoTIFF image a specific geographic location over the Earth surface. Therefore, the regular grid of building pixels in a GeoTIFF image actually represents a 2D grid of locations over the Earth surface, i.e., a 2D spatial sampling *Dimension* in the context of SODA. Thus, *GeoTIFFChannel* is able to import a sampling *Dimension* by simply reading the georeferencing metadata. Since several layers (*rasters*) of data may be recorded in a GeoTIFF image, a single pixel may store observation values of different *observation properties*. Obviously, such data layers providing observation values of a 2D spatial *Dimension* perfectly match the SODA concept of *Extensional Mapping*. Thus, importation/exportation of spatial *Extensional MappingSets* is a straightforward task for *GeoTIFFChannel*. Since a GeoTIFF actually stores georeferenced data, both exporting *Dimensions* and importing/exporting *Constants* are not supported for this *data channel*.

The second *data channel* implemented in the prototype, *PostGISChannel*, enables the importing/exporting of observation data from/to a PostGIS database. The concept of table in relational databases has also a straightforward match to the concept of *Extensional MappingSet*. Therefore, any *Dimension* may be imported/exported from/to a column in a PostGIS table. The correct formatting and semantics of the imported column is under the user sole responsibility. *PostGISChannel* only ensures that no duplicated values are imported. Likewise, an *Extensional MappingSet* may be imported/exported from/to a table in PostGIS, where each column corresponds to either a *Dimension* or an *Extensional Mapping*. Notice that importing/exporting Constants does not make sense in the scope of this *data channel*.

5.6 Operators Implementation

Each operation defined in Section 4.3.3 is implemented by a relevant method in terms of Spark operations on DataFrames. Implementation descriptions for each *Dimension* and *Extensional MappingSet* operation are given in the following sections. Implementation of *Constant* operations is either trivial or very similar to relevant *Extensional MappingSet* operations, thus they are not explicitly explained here.

5.6.1 Dimension Operators

As already explained in Section 5.3.1, each *Dimension* operator is implemented by a relevant method in class *Dimension*. Thus, syntax example and a detailed explanation of the implementation of each such method are provided below.

ImportDimension

Example:

```
Dimension dim_1 = Dimension.Import(Name, channelName, storageName);
```

The static method *Import* of class *Dimension* (Fig. 5.7) is called to execute this operator. First, parameter *channelName* is used to get the name of the Java class that implements the appropriate *DataChannelInterface* for the corresponding *data channel*. *Data Channel Metadata* is overseen by *Catalog Manager* to properly configure *data channels* (Fig. 5.1). The relation between the name of a *data channel* and the name of the Java class that provides the implementation appropriate for such *data channel* is recorded in *Data Channel Metadata*. Once the Java class is obtained, a new object is instantiated and the method *ImportDimension* of *DataChannelInterface* is called. Each data channel must implement this method in a proper way using parameter *storageName* and return a *Dimension* object named *Name*.

ScanDimension

Example:

```
Dimension dim_2 = Dimension.Scan(Name);
```

The Spark operator *range* is used in static method *Scan* of class *Dimension* to generate a DataFrame recording a series of integers from 0 to the size s of the stored *Dimension Name*, i.e., generates a in-memory non-materialized stored *Dimension*. Notice that s is obtained from the local catalog. The number n of partitions is system defined. Spreading references across partitions is the partition strategy predefined for the Spark operator *range*. The steps followed by Spark to generate a distributed DataFrame of references are detailed next:

1. Create an array of integers from 0 to $n-1$.
2. Create a DataFrame with n partitions by parallelizing the previous array, i.e., spread each integer i within the array to a specific partition.
3. Each partition generates a portion of consecutive references following the steps below.
   a) Calculate the minimum reference, $min = \lfloor (i \cdot s)/n \rfloor$.
   b) Calculate the maximum reference, $max = \lfloor ((i+1) \cdot s)/n \rfloor - 1$.

c) Generate the intermediate references from *min* to *max*, $\{ i \mid i \in \mathbb{Z} \,;\, i \in [min, max]\}$.

For example, let $n = 5$ and $s = 17$. First, the following array a is generated: $[0,1,2,3,4]$. Then, a DataFrame with 5 partitions is generated by parallelizing a. Thus, each integer within a is delivered to a different partition. For illustration purposes, let us assume that partition 0 stores integer 0, partition 1 stores integer 1, and so on. Finally, each partition generates a portion of the 17 required references applying the above steps:

– partition 0:

$$min = \lfloor (i \cdot s)/n \rfloor = \lfloor (0 \cdot 17)/5 \rfloor = 0$$
$$max = \lfloor ((i+1) \cdot s)/n \rfloor - 1 = \lfloor ((0+1) \cdot 17)/5 \rfloor - 1 = 2$$
$$\{ i \mid i \in \mathbb{Z} \,;\, i \in [min, max]\} = \{ i \mid i \in \mathbb{Z} \,;\, i \in [0,2]\} = \{0,1,2\}$$

– partition 1:

$$min = \lfloor (i \cdot s)/n \rfloor = \lfloor (1 \cdot 17)/5 \rfloor = 3$$
$$max = \lfloor ((i+1) \cdot s)/n \rfloor - 1 = \lfloor ((1+1) \cdot 17)/5 \rfloor - 1 = 5$$
$$\{ i \mid i \in \mathbb{Z} \,;\, i \in [min, max]\} = \{ i \mid i \in \mathbb{Z} \,;\, i \in [3,5]\} = \{3,4,5\}$$

– partition 2:

$$min = \lfloor (i \cdot s)/n \rfloor = \lfloor (2 \cdot 17)/5 \rfloor = 6$$
$$max = \lfloor ((i+1) \cdot s)/n \rfloor - 1 = \lfloor ((2+1) \cdot 17)/5 \rfloor - 1 = 9$$
$$\{ i \mid i \in \mathbb{Z} \,;\, i \in [min, max]\} = \{ i \mid i \in \mathbb{Z} \,;\, i \in [6,9]\} = \{6,7,8,9\}$$

– partition 3:

$$min = \lfloor (i \cdot s)/n \rfloor = \lfloor (3 \cdot 17)/5 \rfloor = 10$$
$$max = \lfloor ((i+1) \cdot s)/n \rfloor - 1 = \lfloor ((3+1) \cdot 17)/5 \rfloor - 1 = 12$$
$$\{ i \mid i \in \mathbb{Z} \,;\, i \in [min, max]\} = \{ i \mid i \in \mathbb{Z} \,;\, i \in [10,12]\} = \{10,11,12\}$$

– partition 4:

$$min = \lfloor (i \cdot s)/n \rfloor = \lfloor (4 \cdot 17)/5 \rfloor = 13$$
$$max = \lfloor ((i+1) \cdot s)/n \rfloor - 1 = \lfloor ((4+1) \cdot 17)/5 \rfloor - 1 = 16$$
$$\{ i \mid i \in \mathbb{Z} \,;\, i \in [min, max]\} = \{ i \mid i \in \mathbb{Z} \,;\, i \in [13,16]\} = \{13,14,15,16\}$$

SamplingDimension

Example:

```
Dimension dim_3 = Dimension.SamplingDimension(Name, k1, k2);
```

For 1D *Sampling* data types (temporal and *Point1D*), the Spark operator *range* is also used in static method *SamplingDimension* of class *Dimension* to generate a DataFrame with the series of integer values of the result dimension, starting in the initial integer obtained from the first *Constant k1* and finishing in the last integer obtained from the second *Constant k2*. Then, integer values are transformed to adequate *MapalValue* UDTs[5] in the DataFrame of resulting *Dimension*.

For data type *Point2D*, method *SamplingDimension* generates two DataFrames. One DataFrame stores the integer x coordinates of feasible *Point2DValues* from the integer x coordinate of *k1* to the integer x coordinate of *k2*. The other DataFrame stores the integer y coordinates from the integer y coordinate of *k1* to the integer y coordinate of *k2*. Then, a Cartesian product is applied to previous integer coordinates in order to build a 2D rectangular grid of points. Finally, integer coordinates of resulting points are used to generate *Point2DValueUDT* values in the DataFrame of resulting *Dimension*.

Notice that both *Constants* must be of compatible data types to enable an *implicit* casting if needed. Otherwise, an exception is thrown.

Union

Example:

```
Dimension dim_4 = d1.Union(d2);
```

Method *Union* in class *Dimension* implements the *Union* of two *Dimensions* as defined in Section 4.3.1. Let d_1 be the *Dimension* that calls the method *Union* and d_2 be the *Dimension* passed as input argument. Notice that the result of this operation is always a materialized non-stored *Dimension*.

The algorithm to implement the *Union* of two *Dimensions*, where at least one of them is a *sampling Dimension*, is shown in Algorithm 5.1. First, required metadata is obtained from each *Dimension* in Steps 1 - 6. The name of each *Dimension*, required to build the

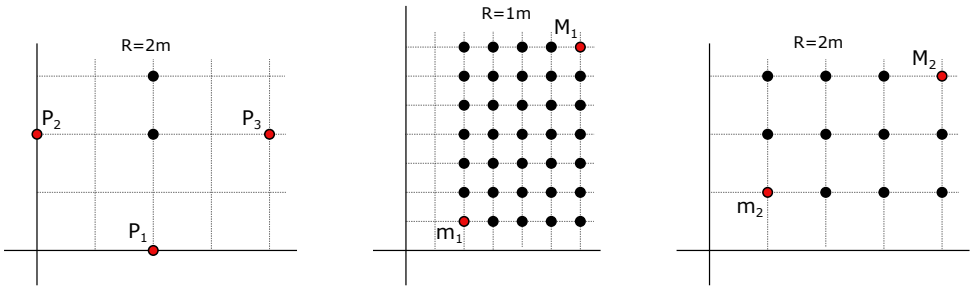
[5] Resulting data type is determined by the data type of *Constants*.

Algorithm 5.1 *Union* algorithm (at least one *Dimension* is a *Sampling*).

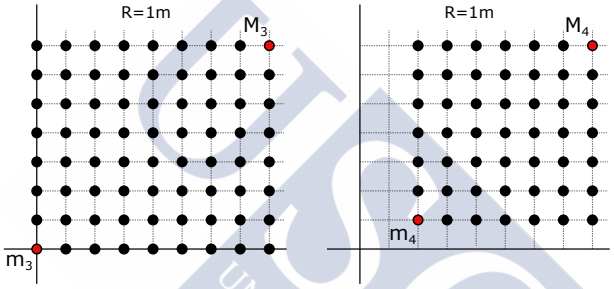
1: $name_1 \leftarrow$ name of d_1
2: $name_2 \leftarrow$ name of d_2
3: $m_1 \leftarrow$ minimum value of d_1
4: $M_1 \leftarrow$ maximum value of d_1
5: $m_2 \leftarrow$ minimum value of d_2
6: $M_2 \leftarrow$ maximum value of d_2
7: Apply an implicit casting to above *SamplingValues* (if needed)
8: **if** $m_1.lowerThanOrEqualTo(m_2)$ **then**
9: $start \leftarrow m_1$
10: **else**
11: $start \leftarrow m_2$
12: **end if**
13: **if** $M_1.greaterThanOrEqualTo(M_2)$ **then**
14: $end \leftarrow M_1$
15: **else**
16: $end \leftarrow M_2$
17: **end if**
18: $name =$"Union_ $name_1$_$name_2$"
19: **return** $Dimension.SamplingDimension(name, start, end)$

name of the resulting *Dimension*, is always obtained from the header. The minimum and maximum values of each *Dimension* are required to calculate the start and end values of the resulting *sampling Dimension*. For *sampling Dimensions*, minimum and maximum values are obtained from start and end values stored in the header. For non sampling *Dimensions*, data values are first obtained by materialization, and then, minimum and maximum values are calculated by applying the aggregate DataFrame methods *min* and *max* to previous data values. Notice that minimum and maximum values are of a subtype of *SamplingValue* and must be of compatible data types for the operation *Union* in order to enable an *implicit* casting if needed in Step 7. Start and end values of resulting *Dimension* are calculated in Steps 8 - 17 by calling appropriate comparison methods. A special case arises if *Dimension* values are of type *Point2DValue*. If both *Dimensions* are *sampling Dimensions*, the coordinates of start and end values are calculated as follows.

$$start = (\min(m_1.getX(), m_2.getX()), \min(m_1.getY(), m_2.getY()))$$
$$end = (\max(M_1.getX(), M_2.getX()), \max(M_1.getY(), M_2.getY()))$$



(a) Non sampling Dimension D1 (b) Sampling Dimension D2 (c) Sampling Dimension D3



(d) Union(D1,D2) (e) Union(D2,D3)

Figure 5.11: Examples of operation Union with sampling Dimensions.

If only one *Dimension*, lets say d_1, is a *sampling Dimension*, the start and end values are calculated as shown below,

$$start = (\min(m_1.getX(), \min(x_1, \ldots, x_n)), \min(m_1.getY(), m_2.getY())),$$
$$end = (\max(M_1.getX(), \max(x_1, \ldots, x_n)), \max(M_1.getY(), M_2.getY())),$$

where $x_1, \ldots, x_n$ are the x coordinates of values in the non sampling *Dimension* d_2. Step 18 derives a new name for the result *sampling Dimension*, which is constructed and returned in Step 19. Examples of such result *sampling Dimensions* are depicted in Fig. 5.11. The *sampling Dimension* resulting from the *Union* of a non sampling *Dimension D1* (Fig. 5.11(a)) and a *sampling Dimension D2* (Fig. 5.11(b)) is depicted in Fig. 5.11(d). Notice that the start value m_3 of the result *Dimension* is built with the x coordinate of P_2 and the y coordinate of P_1, applying the above formulas. Likewise, the end value of the result *Dimension* is built with the x coordinate of P_3 and the y coordinate of M_1. Fig 5.11(e) shows the *sampling Dimension* resulting from the *Union* of *sampling Dimensions D2* and *D3* depicted in Fig. 5.11(b) and

Algorithm 5.2 *Union* algorithm (both *Dimensions* are non sampling *Dimensions*).

1: $d_1.Materialize()$ (if needed)
2: $d_2.Materialize()$ (if needed)
3: Apply an implicit casting to data values of both Dimensions (if needed)
4: $D_1 \leftarrow$ DataFrame of d_1
5: $D_2 \leftarrow$ DataFrame of d_2
6: $D \leftarrow D_1.UnionAll(D_2).dropDuplicates()$
7: $H \leftarrow$ generate new header
8: **return** $Dimension(H, D)$

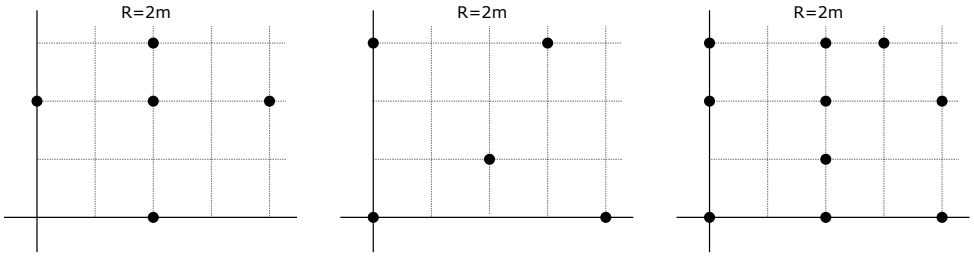
Fig. 5.11(c) respectively. Start value $m4$ of the result *Dimension* is built with the x coordinate of m_2 and the y coordinate of m_1, whereas the end value M_4 is built with the x coordinate of M_2 and the y coordinate of M_1.

The algorithm implemented to calculate the *Union* of two non sampling *Dimensions* is shown in Algorithm 5.2. First, input *Dimensions* are materialized if needed in Steps 1 - 2. Then, an optional *implicit* casting may be applied if required in Step 3. Operation *Union* is performed in Steps 4 - 6 by calling the DataFrame method *UnionAll* over the DataFrames of d_1 and d_2. Duplicate data values must be deleted from the DataFrame D of the resulting *Dimension*. Step 7 generates the header H of the resulting Dimension. Finally, a new *Dimension*, generated from H and D, is returned. Notice that the returned *Dimension* is a non sampling *Dimension*.

An example of the *Union* of two non sampling *Dimensions* is shown if Fig. 5.12. The non sampling *Dimension* resulting from the *Union* of non sampling *Dimensions D1* and *D4*, Fig. 5.12(a) and Fig. 5.13(a) respectively, is depicted in Fig. 5.12(c). The resulting *Dimension* contains all the values recorded both in *D1* and *D4*.

As stated in Section 5.3.2, a two-column DataFrame is stored per non sampling *Dimension* in the local catalog to record data and references, whereas only metadata storage is required for *sampling Dimensions*. Therefore, different materialization strategies are required for each *Dimension* type. The materialization process of a non sampling *Dimension* d is a simple process involving the following two steps:

1. Read from local catalog the stored *Dimension* corresponding to d, i.e., read from disk both the header metadata and data values.



(a) Non sampling Dimension D1 (b) Non sampling Dimension D4 (c) Union(D1,D4)

Figure 5.12: Examples of operation Union with non sampling Dimensions.

2. Let D_1 be the DataFrame storing the references of d. Let D_2 be the two-column DataFrame of the stored *Dimension* read from local catalog. Data values from D_2 matching the references in D_1 can be obtained by applying a $left\ join$ by reference columns over D_1 and D_2.

For a *sampling Dimension d*, the materialization process is even simpler:

1. Read from local catalog the stored *Dimension* corresponding to d, i.e., read from disk the header metadata.

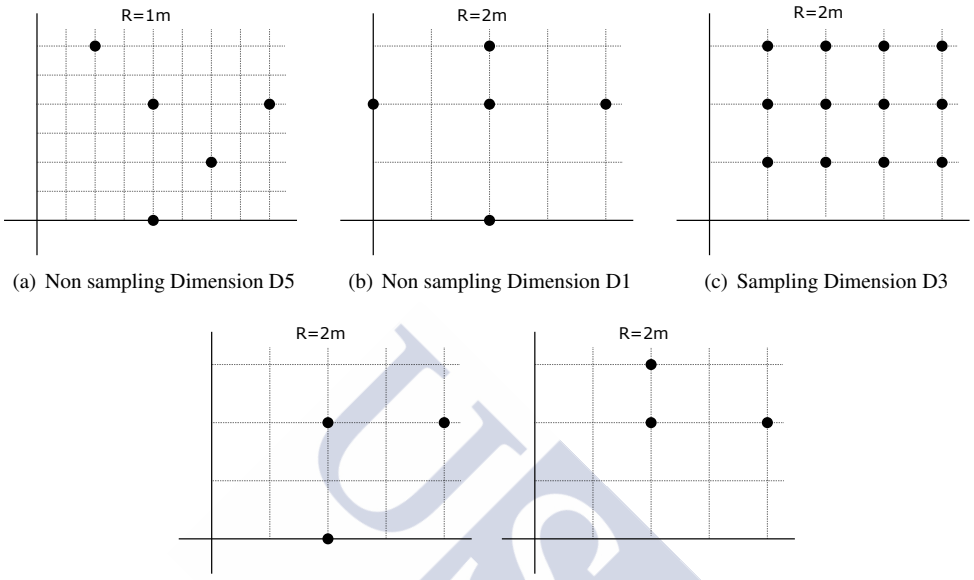
2. Use the minimum and the maximum values obtained from metadata with the Spark operator *range* to generate a DataFrame storing the references to resulting *Dimension* values. Then, the Spark operator *map* is used to generate the resulting *SamplingValues* from relevant references.

Intersection

Example:

```
Dimension dim_5 = d1.Intersection(d2);
```

The algorithm for the *Intersection* of two *Dimensions*, defined in Section 4.3.1, is implemented by method *Intersection* of class *Dimension*. Let d_1 be the *Dimension* that calls the method *Intersection* and d_2 be the *Dimension* passed as input argument. Similarly to operator *union*, the result of this operation is always a materialized non-stored *Dimension*.



(a) Non sampling Dimension D5 (b) Non sampling Dimension D1 (c) Sampling Dimension D3

(d) Intersection(D1,D5) (e) Intersection(D1,D3)

Figure 5.13: Examples of operation Intersection with non sampling Dimensions.

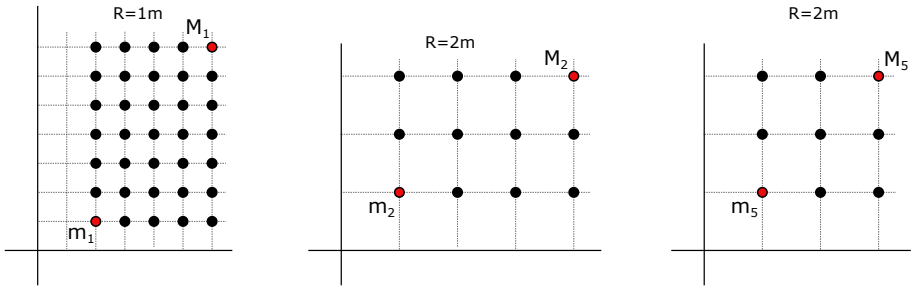
The *Intersection* of two *Dimensions*, where at least one of them is a non sampling *Dimension*, can be implemented following an algorithm similar to that shown in Algorithm 5.2 by simply changing Step 6 by the line below, which calls the DataFrame method *Intersect*.

$$D \leftarrow D_1.Instects(D_2).dropDuplicates()$$

Examples of operation *Intersection* are provided in Fig. 5.13. The non sampling *Dimension* resulting from the *Intersection* between the non sampling *Dimensions D1* and *D5* (Fig. 5.13(b) and Fig. 5.13(a) respectively) is depicted in Fig. 5.13(d). The *Intersection* of the non sampling *Dimension* D1 and the *sampling Dimension D3* (Fig. 5.13(c)) results in the non sampling *Dimension* shown in Fig. 5.13(e).

If both *Dimensions* are 1D *Samplings*, then the algorithm to implement the Intersection is similar to that shown in Algorithm 5.1. Condition in Step 8 is changed to $m_1.greaterThanOrEqualTo(m_2)$, condition in Step 13 changes to $M_1.lowerThanOrEqualTo(M_2)$, and the resulting name generated in Step 18 is "Intersection_ $name_1$_$name_2$".

Similarly to operation *Union*, the intersection of two 2D *Samplings* can be calculated by Algorithm 5.1 with small changes in the computation of start and end values of resulting



(a) Sampling Dimension D2 (b) Sampling Dimension D3 (c) Intersection(D2,D3)

Figure 5.14: Examples of operation Intersection with sampling Dimensions.

sampling Dimension. In addition to previous changes in Steps 8, 13 and 18, start and and values are calculated as follows.

$$start = (\max(m_1.getX(), m_2.getX()), \max(m_1.getY(), m_2.getY()))$$
$$end = (\min(M_1.getX(), M_2.getX()), \min(M_1.getY(), M_2.getY()))$$

Fig. 5.14 depicts the *Intersection* between the two *sampling Dimensions D2* and *D3*. Notice that value m_5 in resulting *sampling Dimension* is built with x coordinate of m_1 and y coordinate of m_2. Likewise, value M_5 is built with x coordinate of M_1 and y coordinate of M_2.

ProjectDimension

Example:

```
Dimension dim_6 = Dimension.Project(d, c, MS);
```

Operation *ProjectDimension* is implemented by static method *Project* of class *Dimension*. A preliminary filtering operation is applied to specified *Extensional MappingSet MS* depending on whether parameter *c* (pointing to a boolean *Extensional Mapping* of *MS*) is provided or not. If provided, DataFrame operator *filter* is used to restrict to the rows where *c* is `true`. Next, DataFrame operator *select* is evaluated to generate one *Dimension* from the *Dimension* or *Extensional Mapping* pointed by parameter *d*. Finally, duplicates are removed using DataFrame operator *dropDuplicates*. Notice that duplicates have to be eliminated regardless of whether the *Dimension* is materialized or not.

StoreDimension

Example:

```
dim_6.Store(Name);
```

Method *Store* of class *Dimension* persists an in-memory *Dimension* to disk. The name of the stored *Dimension* in the local catalog is given by parameter *Name*. Relevant metadata is recorded into local catalog, including minimum and maximum values for *sampling Dimensions*. Additionally, as explained in Section 5.3.2, data values of non sampling *Dimensions* are stored in two-column Parquet files. First, non-materialized *Dimensions* must be materialized. Then, a new DataFrame recording a column with data values of *Dimension* in ascending order is generated. Next, a new references column is added to the DataFrame. Finally, the two-column DataFrame is stored in a Parquet file. Storing references column in disk enables future lazy materialization.

ExportDimension

Example:

```
dim_6.Export(channelName, storageName);
```

Method *Export* of class *Dimension* is similar to method *Import*. First, parameter *channelName* is used to get the name of the Java class that implements the appropriate *DataChannelInterface* for the corresponding *data channel*. Once the Java class is obtained, a new object is instantiated and the method *ExportDimension* of *DataChannelInterface* is called to store the *Dimension* in the *data channel* named as *storageName*. Each *data channel* must implement this method in a proper way.

5.6.2 Extensional MappingSet Operators

Each *MappingSet* operator is implemented by a relevant method in class *MappingSet*, as stated in Section 5.3.1. The rest of this section provides a syntax example and a broad description of the implementation of each method.

ImportMappingSet

Example:

```
MappingSet ms_1 = MappingSet.Import(Name, channelName, storageName, domain);
```

Method *Import* of class *MappingSet*, Fig. 5.7, enables an *Extensional MappingSet* to be imported from an external *data channel*. The name of the Java class implementing the *DataChannelInterface* for the corresponding *data channel* is obtained from parameter *channelName*. Then, a new object is instantiated and method *ImportMappingSet* is called. Similarly to *Dimensions*, each *data channel* must implement this method.

Product

Example:

```
MappingSet MS1 = MappingSet.Product(Name, dims);
MappingSet MS2 = MS1.Product(dim);
```

Two overloaded methods *Product* have been defined in class *MappingSet* to implement this operator. The static method that accepts a list of *Dimensions* as input parameter uses the DataFrame operator *join*[6] over input *Dimensions* to build the domain of the returned *Extensional MappingSet*. The method that accepts only one *Dimension* as input parameter uses the DataFrame operator *join* over the input *Dimension* and the calling *Extensional MappingSet* to build the returned *Extensional MappingSet*. Notice that the resulting domain is the Cartesian product of the input *Dimension* and the current domain of calling *Extensional MappingSet*, and all *Extensional Mappings* are kept in the resulting *Extensional MappingSet*.

ProjectMappingSet

Example:

```
MappingSet MS3 = MS2.Project(name, components);
```

This operation is implemented by method *Project* of class *MappingSet*. DataFrame operator *select* is used to get the resulting DataFrame with the domain of the calling *Extensional*

[6] Join without conditions is equivalent to Cartesian product.

MappingSet and only the *Dimensions* and *Extensional Mappings* referenced by input argument *components*. Notice that non-materialized *Dimensions* referenced by *components* must be materialized before applying the operator *select*.

EvaluateIntensionalMappings

Example:

```
MappingSet MS3 = MS2.EvaluateIntensionalMappings(mappingNames, mappings);
```

Evaluation of *Intensional Mappings* is implemented by method *EvaluateIntesionalMappings* of class *MappingSet*. A SQL-like expression is generated for each *intensional* mapping call in input argument *mappings*. Thus, DataFrame operator *selectExpr* can be used to evaluate them and generate one new column per *intensional* mapping expression. The name of each new generated mapping is provided by input argument *mappingNames*. Notice that *primitive* mappings must be available as Spark UDFs in order to be called by operator *selectExpr*.

EvaluateExtensionalMapping

Example:

```
MappingSet MS3 = MS.EvaluateExtensionalMapping(mappingName, mapping);
```

Method *EvaluateExtensionalMapping* of class *MappingSet* implements this operator as shown in Algorithm 5.3. Notice that input parameter *mapping* stores an expression of the form $ems.em(s_1, \cdots, s_m)$, where *ems* references an *ExtensionalMappinSet*, *em* references an *ExtensionalMapping* of *ems*, and each s_i is the name of either a *Dimension* or an *Extensional Mapping* of the calling *Extensional MappingSet MS*. Let $em(d_1, \cdots, d_m)$ denote the *Extensional Mapping* to be evaluated. Before *em* is obtained from disk, each s_i must record in *MS* references to its relevant d_i. Four cases have been identified:

– Reference s_i is the name of an *Extensional Mapping* of *MS*. If d_i is in the list of *Dimensions* referenced by s_i, then an appropriate column with references to d_i is already prepared to evaluate *em*. Otherwise, references to d_i must be obtained form values stored in s_i (function *createRefsToStoredDim* in Algorithm 5.3). First, *Dimension* d_i is obtained from disk and materialized. Then, an *inner equi-join* is evaluated between DataFrames of d_i and *MS* using value columns of d_i and s_i in the equality condition to

Algorithm 5.3 *Evaluate Extensional Mapping* algorithm.

```
1:  D ← DataFrame of MS
2:  for i ← 1 to n do
3:      if isExtensionalMapping(s_i) then
4:          if d_i ∉ referencedBy(s_i) then
5:              D ← createRefsToStoredDim(D, s_i, d_i)
6:          end if
7:      else
8:          if isDimension(s_i) then
9:              if storageName(s_i) != d_i then
10:                 s_i.Materialize() (if needed)
11:                 D ← createRefsToStoredDim(D, s_i, d_i)
12:             end if
13:         else
14:             return Error
15:         end if
16:     end if
17: end for
18: H ← update MS header
19: D ← materializeMapping(ems, em, D)
20: return ExtensionalMappingSet(H, D)
```

get the relevant d_i reference corresponding to each s_i value. Finally, the value column of d_i is dropped from result.

– Reference s_i is the name of a *Dimension* of *MS*. If its storage name is equal to the name of *Dimension* d_i, then *MS* already has a column with references to d_i. Otherwise, references to d_i must be obtained form values stored in s_i as in the previous case. Notice that s_i must be materialized before references are obtained.

Once all required reference columns have been obtained, function *materializeMapping* in Algorithm 5.3 materializes *ems.em*. Recall that, as explained in Section 5.3.2, stored *Extensional MappingSets* record a single column with references to their domain values. Therefore, *ems.em* is obtained from disk as a two column DataFrame $EMDF$, where column $refs$ contains references to the domain of *ems* and column $values$ contains the relevant values of *ems.em*. At this stage, each s_i has a column r_i in *MS* that records references to d_i. Thus, a new column *domainRefs* with relevant references to domain values of *ems* can be obtained from

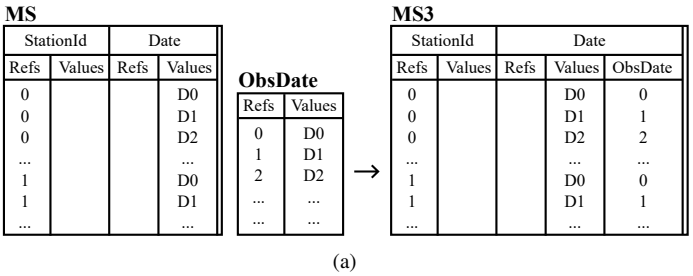


Figure 5.15: Example of operation EvaluateExtensionalMapping.

reference columns r_i in MS through the following expression.

$$dr = r_n \cdot 1 + r_{n-1} \cdot \text{size}(r_n) + r_{n-2} \cdot (\text{size}(r_n) \cdot \text{size}(r_{n-1})) + \ldots + r_1 \cdot (\text{size}(r_n) \cdot \ldots \cdot \text{size}(r_2))$$

Finally, an *equi-join* is executed between the DataFrame of MS and $EMDF$ using column $domainRefs$ and column $refs$ in the equality condition. Notice that columns $domainRefs$ and $refs$ are dropped from the resulting DataFrame, whereas columns r_i are kept for subsequent operations.

An illustration example, depicted in Fig. 5.15, evaluates the following sentence:

```
MappingSet MS3 = MS.EvaluateExtensionalMapping("Temperature", "Observation.Temperature(
    StationId,Date)");
```

where *Observation* is a stored *MappingSet* whose domain is composed of stored *Dimensions* *StationId* and *ObsDate*. Hence, $d_1 = StationId$, $d_2 = ObsDate$, $s_1 = StationId$, and $s_2 = Date$.

The execution of operator *EvaluateExtensionalMapping* in this example follows the steps below.

- Fig 5.15(a). Let MS be an *Extensional MappingSet* whose domain is composed of *Dimensions* $StationId$ and $Date$. Since $d_1 = s_1$, $MS.StationId.Refs$ records references to the stored *sampling Dimension* $StationId$ ($r_1 = MS.StationId.Refs$). However, $Date$ is a non-stored *sampling Dimension* recording date values. Thus, we have to build the appropriate references to *Dimension* $ObsDate$ from such values. First, *Dimension* $ObsDate$ is read from disk. Then, an *equijoin* operation is executed between MS and $ObsDate$ with the following condition $MS.Date.Values = ObsDate.Values$ to assign each $Date$ value the proper reference to $ObsDate$. The new generated references column is stored as $MS.Date.ObsDate$ ($r_2 = MS.Date.ObsDate$) to build an intermediate result of $MS3$.
- Fig. 5.15(b). Once we have all required reference columns r_i, column $MS3.DomainRefs$ is built applying the above formula.
- Fig. 5.15(c). In the last step, the complete *Extensional Mapping* to be evaluated is first read from disk together with references to the domain of the relevant *ExtensionalMappingSet* ($EMDF$). An *equijoin* is executed between the intermediate result of $MS3$ and $EMDF$ with the following condition $MS3.DomainRefs = EMDF.Refs$ to restrict to temperature values referenced by $MS3$.

As we can see, the new *Extensional MappingSet MS3* is composed of the same domain as *Extensional MappingSet MS* and the new *Extensional Mapping Temperature*. Notice that column $MS.Date.ObsDate$ remains for further operations.

EvaluateConstant

Example:

```
MappingSet MS4 = MS.EvaluateConstant(mappingName, k);
```

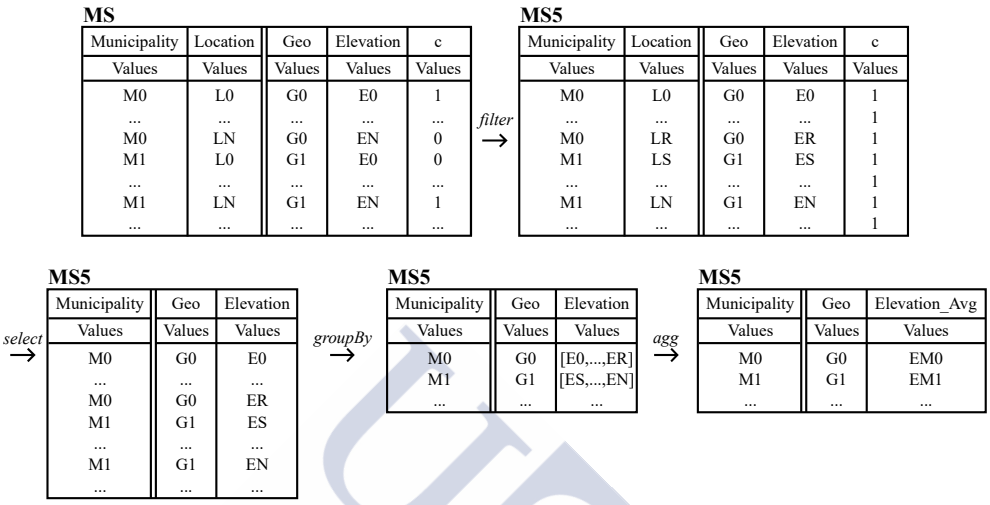


Figure 5.16: Example of operation EvaluateAggregateMapping.

DataFrame operator *withColumn* is used by method *EvaluateConstant* in class *MappingSet* to add a new column *mappingName* to the calling *Extensional MappingSet MS*. A new column containing the same *MapalValue* in all rows is generated from input *Constant k* and passed as parameter to operator *withColumn* to be added to the DataFrame of *MS*.

EvaluateAggregateMappings

Example:

```
MappingSet MS5 = MS.EvaluateAggregateMappings(name, groupBy, orderBy, c, mappingNames,
     aggs);
```

Implementation of this operator in method *EvaluateAggregatedMappings* of class *MappingSet* is as follows. First, the DataFrame operator *filter* with condition $c =$ `TRUE` is applied to calling *Extensional MappingSet MS* in order to select the required rows. Next, DataFrame operator *select* is used to drop from *MS* those *Dimensions* not referenced in input argument *groupBy* and those *Extensional Mappings* whose domain contains *Dimensions* not present in input argument *groupBy* (except those referenced by aggregated mappings). Notice that non-materialized *Dimensions* referenced by aggregated mappings must be materialized. Then, DataFrame operator *groupBy* prepares the DataFrame of *MS* for the subsequent application of

relevant aggregate mappings by using DataFrame operator *agg*. Appropriate aggregate mappings, implemented as UDFs, are used by this operator. Finally, DataFame operator *orderBy* is used to order the resulting *Extensional MappingSet* according to input argument *orderBy*.

In Fig. 5.16, an illustration example is shown. Let *MS* be a *Extensional MappingSet* whose domain is composed of *Dimensions Municipality* and *Location*, recording municipality ids and a 2D sampling of georeferenced points, respectively. Additionally, three *Extensional Mappings* have been defined in *MS*. *Geo* records the relevant geometry of each municipality. *Elevation* records the elevation above the sea level of each location point. *c* is a boolean *Extensional Mapping* taking true values for those tuples where *Location* is within *Geo*. The average elevation for each municipality is calculated in this example by executing the following sentence:

```
MappingSet MS5 = MS.EvaluateAggregatedMappings("MS5", "Municipality", "Municipality", c
    "Elevation_Avg", aggs);
```

where *aggs* is an *AggMappingCall* representing the expression AVG(*Elevation*).

The first step in Fig. 5.16 shows how *MS* is filtered with the condition $c = \texttt{TRUE}$ to obtain the first intermediate result of *MS*5. Then, *Dimensions* not referenced in argument *groupBy* (*Location*) and *Extensional Mappings* whose domain contains *Dimensions* not referenced in argument *groupBy* (*c*) are removed. Next, DataFrame operator *groupBy* is used to group by *Dimension Municipality* and *Extensional Mapping Geo*. The aggregated mappings are now evaluated over the referenced *Extensional Mapping Temperature* to generate the new aggregated mapping column *Elevation_Avg*. Notice that those *Dimensions* and *Extensional Mappings* referenced by aggregated mappings that do not meet the select condition (*Temperature*) are now removed. Finally, the resulting Extensional MappingSet is ordered by *Dimension Municipality*.

ExportMappingSet

Example:

```
MS.Export(storageName, channelName);
```

Similarly to method *Export* in class *Dimension*, parameter *channelName* is used to get the name of the Java class that implements the appropriate *DataChannelInterface* for the corresponding *data channel*. Once the Java class is obtained, a new object is instantiated and

the method *ExportMappingSet* in *DataChannelInterface* is called. Each data channel must implement this method in a proper way.

StoreMappingSet

Example:

```
MS.Store(storageName);
```

Storage of *Extensional MappingSets* into the local catalog is performed by method *Store* in class *MappingSet*. First, DataFrame of calling *Extensional MappingSet MS* is ordered by reference columns r_i of *Dimensions* using DataFrame operator *sort*. To ensure that every *Dimension* has a relevant reference column r_i, all *Dimensions* within *MS* must be previously stored into the local catalog. Next, DataFrame operation *select* is used to select the *Extensional Mapping* columns of *MS* and to generate a single column with references to the domain of *MS* by applying the expression defined to build the column *domainRefs* during the evaluation of *Extensional Mappings*. The resulting DataFrame is recorded in a Parquet file. Additionally, *MappingSets* and *Mappings* structures of local catalog (as shown in catalog example of Fig. 5.9) are updated with appropriate metadata.

5.7 Experimental Evaluation

5.7.1 Cluster setup

All experiments were conducted on the Big Data cluster at Centro de Supercomputación de Galicia (CESGA)[25], consisting of 38 nodes with two configurations:

- 4 master nodes
  - 2x 6-core Intel Xeon E5-2620 v3 @ 2.40GHz
  - 64 GB RAM
  - 1x 10Gbps + 2x 1Gbps
  - 12x 2TB NL SATA 6Gbps 3.5”

- 34 slave nodes
  - 2x 6-core Intel Xeon E5-2620 v3 @ 2.40GHz
  - 64 GB RAM
  - 1x 10Gbps + 2x 1Gbps
  - 8x 480GB SSD SATA 2.5”

Each node runs CentOS Linux release 7.4.1708 with Hadoop 2.4.2, Spark 1.6.1 and Spark 2.1.2. The Spark cluster is deployed on YARN (Yet Another Resource Navigator) [12].

5.7.2 Experiment setup

A great number of criteria can be applied to classify *queries* in database management systems (DBMS). One of them is based on the number of *scans* required to access an object. According to this criterion, we may have either *single-scan queries* (i.e., require at most one access to an object) or *multiple-scan queries* (objects have to be accessed several times). Obviously, *multiple-scan queries* are more time consuming because execution time is generally not linear but superlinear in the number of objects [21]. One of the most important *multiple-scan query* in a spatial DBMS is the *spatial join*, defined in [86] as follows.

> Given two relations, R and S, each storing a set of spatial objects (i.e., each tuple stores the identifier of one object), spatial join identifies overlapping object from R and S. (A range query is a special case in which one of the relations represents the set of points and the other relations represents the query region.) The spatial join is denoted by
>
> $$R[zr \diamond zs]S$$
>
> where zr and zs are the attributes of R and S (respectively) that store the elements resulting from the decomposition of spatial objects.
>
> (Orenstein, 1986: 329)

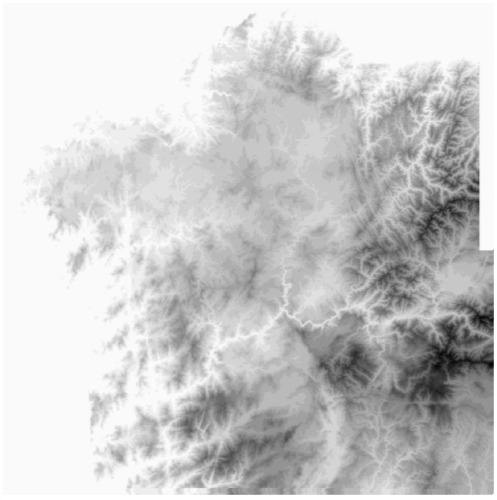
Based on the above, a *spatial join* operation with a *contains* predicate is used to compare performance between existing solutions. Thus, the execution time of *spatial join* is used as comparison attribute. Furthermore, to test the integration capabilities of MAPAL, a *raster* of *Point2D* values is joined with a set of polygons (entities).

Spatial Resolution (meters)	Size (# points)
400	391732
200	1564376
100	6252500
50	2500000
25	9998000
12.5	399880000

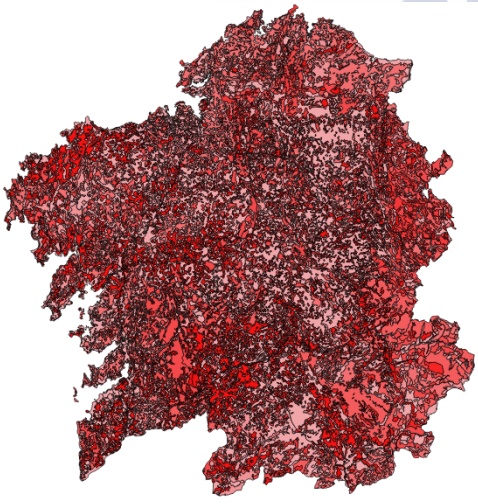
Table 5.1: Spatial resolution and number of points of *raster* datasets.

For this experiment, elevation data of Galician geography at different spatial resolutions provided by the Spanish National Geographic Institute (Instituto Geográfico Nacional - IGN) [57] is used as input *raster* data, whereas a combustion model layer obtained from land cover classification of Galicia is used as input *entity* data. Spatial resolutions (in meters) of *raster* datasets and relevant size (number of points) are shown in Table 5.1. Fig. 5.17(a) shows the elevation *raster* at a spatial resolution of 200 meters. Fig 5.17(b) depicts the combustion model dataset, which is composed of 11057 polygons. The expected output joins each polygon P of the combustion model dataset with all locations of the elevation *raster* contained in P. However, as explained in Section 5.3.1, the *in-memory* MappingSet structure defined in MAPAL requires each domain value to be composed of individual *Dimension* values. Thus, the output returned by MAPAL contains all the elements of the Cartesian product of polygons and *raster* points. An additional boolean mapping returns `true` if the relevant polygon contains the relevant point and returns `false` otherwise. The sequence of MAPAL operators generated to evaluate the *spatial join* between the combustion model polygons *CombustionPolygons* and the *raster Raster_25meters* with a spatial resolution of 25 meters is shown below for illustration purposes.

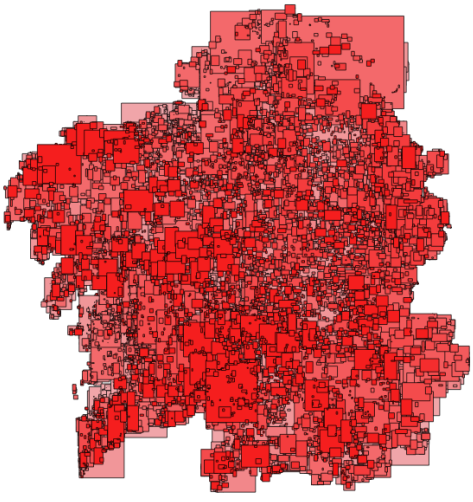
```
Dimension polygonIds = Dimension.Scan("CombustionPolygonIds");
Dimension points25meters = Dimension.Scan("Raster_25meters").Materialize();
MappingSet domain = MappingSet.Product("domain", polygonIds, points25meters);
MappingSet geo = domain.EvaluateExtensionalMapping("Geo", "CombustionPolygons.Geo(
    CombustionPolygonIds)");
MappingSet result = geo.EvaluateIntensionalMappings("Contains", "contains(Geo,
    Raster_25meters)");
```



(a) Elevation raster data



(b) Combustion Model Polygons



(c) Combustion Model Envelopes

Figure 5.17: Experiment input datasets.

5.7.3 Evaluation Results

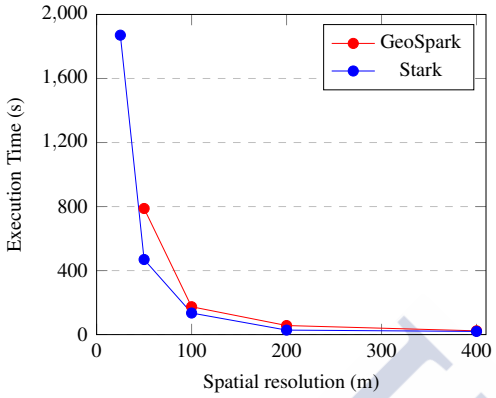
The frameworks selected to be compared against the MAPAL prototype are the most relevant distributed spatial data processing systems in the state of art developed on top of Spark, i.e., GeoSpark [108], Geotrellis [43], LocationSpark [95], Simba [106], SpatialSpark [107] and Stark [92].

To ensure a fair comparison between the different distributed spatial data processing frameworks, the input dataset has been pre-processed. Since some existing frameworks do not support columns of non-spatial data types, additional input columns storing non spatial values (e.g., elevation values column) have not been included in the analysis tasks. Furthermore, since LocationSpark only enables spatial processing of rectangular polygons, each input polygon has been replaced by its *envelope*. Fig. 5.17(c) shows the *envelopes* used in the experiment. Additionally, since the rest of existing solutions do not provide integrated *raster-entity* data analysis, location points within the elevation *raster* are translated to a set of 2D points for each tested solution.

To test the performance of each solution for different workloads, the *spatial join* operation has been executed between the combustion model *envelopes* and the *raster* datasets shown in Table 5.1. Fig. 5.18 shows the *spatial join* execution time (each time value is actually the average of time values provided by five executions) for tested frameworks with the following Spark configuration:

- *master*: yarn
- *deploy-mode*: cluster
- *driver-memory*: 8G
- *executor-memory*: 8G
- *num-executors*: 40

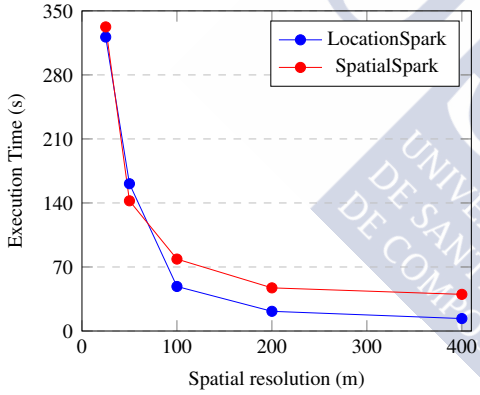
Due to the huge difference in resulting execution times (orders of magnitude) between tested solutions, such results have been plotted in three different charts. Notice that Mapal (Fig. 5.18(e)) is the slowest solution. Almost two times slower than SpatialSpark (Fig. 5.18(c)) at a spatial resolution of 400 meters, and almost one order of magnitude slower than Stark (Fig. 5.18(a)) at a spatial resolution of 12.5 meters. Therefore, the higher the resolution, the more significant the difference will be.



(a)

Spatial Resolution (m)	GeoSpark	Stark
400	23.44	19.83
200	56.60	28.05
100	174.00	134.74
50	787.82	469.12
25		1870.64
12.5		7047.82

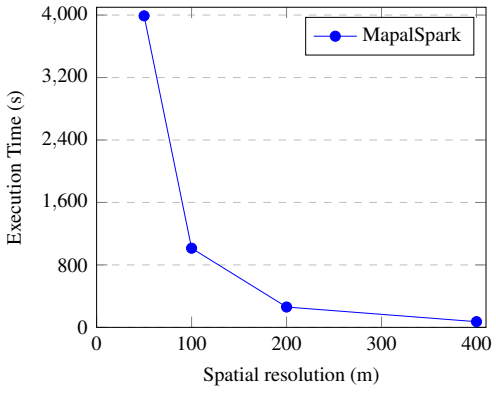
(b)



(c)

Spatial Resolution (m)	LocationSpark	SpatialStark
400	13.61	40.07
200	21.53	47.17
100	48.69	78.65
50	161.03	142.34
25	321.40	332.49
12.5	1169.58	

(d)



(e)

Spatial Resolution (m)	MapalSpark
400	72.98
200	260.76
100	1013.79
50	3990
25	15960
12.5	63405

(f)

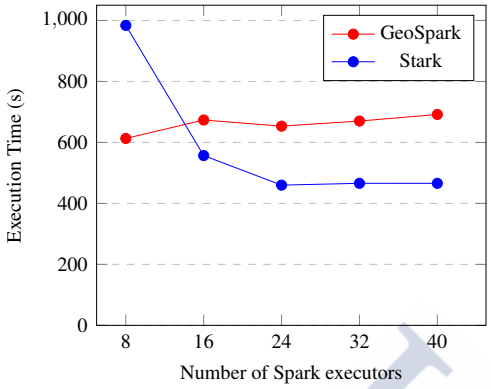
Figure 5.18: Execution time for the *spatial join* operation in a 40-executor cluster.

Some solutions showed Java Heap Memory Overflow as resolution increased, e.g., GeoSpark (25 and 12.5), SpatialSpark (12.5). Simba was tested but it always had to be killed because it seemed to be hanged. Thus, I have got no results for Simba. I was not allowed to execute Geotrellis in a 40-executor configuration due to the huge amount of disk accesses per second. Only a 8-executor configuration was allowed to test Geotrellis. An example of disk accesses on such configuration provided by the hdfs audit log is shown below. The number of accesses increases from 368 accesses/minute to 31818 accesses/minute when Geotrellis starts the *spatial join* operation at 18:27, and decreases from 31931 accesses/minute to 117 accesses/minute when Geotrellis is killed at 18:32.

```
[root@c13-19 hdfs]# for i in {25..35}; do echo "Hour: 18:$i"; head -n 1000000 hdfs-audit
    .log.2018-05-12 | grep "18:$i" | wc -l; done
Hour: 18:25
547
Hour: 18:26
368
Hour: 18:27
31818
Hour: 18:28
31812
Hour: 18:29
31813
Hour: 18:30
31822
Hour: 18:31
31913
Hour: 18:32
117
Hour: 18:33
83
Hour: 18:34
81
```

5.7.4 Scalability

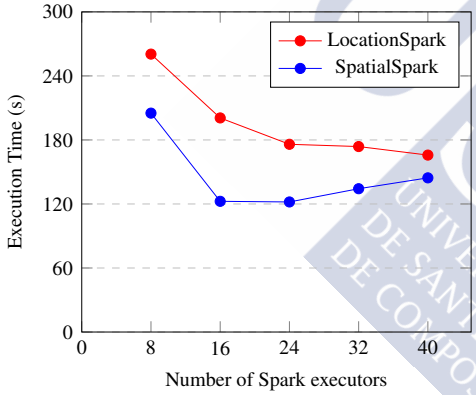
A major feature of every large-scale data processing system is the scalability, i.e., the capability to handle a growing amount work and the potential to be enlarged to accommodate that growth [18]. Therefore, scalability is a main requirement for Mapal. To test the performance of Mapal regarding scalability, the *spatial join* operation has been executed on Spark clusters with different number of executors. In order to enable the execution of all the remainder solutions, a spatial resolution of 50 meters has been selected.



(a)

Number of Spark executors	GeoSpark	Stark
8	613.02	983.87
16	673.49	556.88
24	653.38	459.92
32	670.08	465.90
40	691.70	465.83

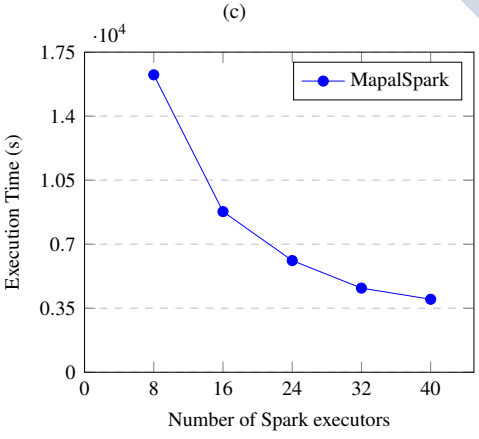
(b)



(c)

Number of Spark executors	LocationSpark	SpatialStark
8	260.43	205.12
16	200.63	122.48
24	175.93	121.89
32	173.81	134.30
40	165.81	144.47

(d)



(e)

Number of Spark executors	MapalSpark
8	16259
16	8782
24	6105
32	4600
40	3990

(f)

Figure 5.19: Execution time for the *spatial join* operation with a spatial resolution of 50 meters.

Similarly to performance charts, scalability results have been plotted in three different charts due to the huge difference in resulting execution times. Again, Mapal (Fig. 5.19(e)) is more than one order of magnitude slower than GeoSpark and Stark (Fig. 5.19(a)), and even slower than LocationSpark and SpatialSpark (Fig. 5.19(c)). Surprisingly, GeoSpark showed a bad scalability performance increasing the execution time as the number of executors increased. Mapal and LocationSpark get to decrease their execution times as the number of executors increased, whereas Stark and SpatialSpark decreased execution times up to 24 executors and then began to increase it. Specifically, Mapal shows a nice scalability behavior.

5.8 Optimization Example

As shown in the previous section, taking into account all the elements of the Cartesian product (e.g., for a spatial resolution of 50 meters, Mapal returns 276175011047 values whereas 36908866 values are returned by remainder solutions) in the evaluation of the primitive mapping *contains* is harmful for the MAPAL performance. Most of the evaluated combinations return `false` in MAPAL, whereas these combinations are not evaluated in the remainder solutions. Therefore, an optimized operator may be build into MAPAL to reduce the huge amount of returned elements while maintaining a *MappingSet* representation consistent with the defined data model.

Two novel *spatial join* operators are proposed here as optimization examples. Both solutions are intended to return a significantly smaller number of elements. The operator *SpatialJoin_Contains_DatatypeQT* must be called from the *MappingSet* of polygons to execute the algorithm shown in Algorithm 5.4. First, in Steps 9 and 10, a Quad-Tree [39] structure covering the whole area of the underlying data type is created for the input *raster* data, generating rectangles that represent groups of input *raster* points. Notice that the values of such data type are ordered following a Z-Order curve [49], thus resulting rectangles are composed of consecutive points. Next, each combustion model polygon is inserted into a copy of the original Quad-Tree structure, which is modified according to the inserted geometry (Steps 13 and 14). Finally, the structure is processed to return an array with the resulting rectangles and additional metadata indicating whether the points of each rectangle are contained in the relevant polygon or not (Step 15). Thus, the values of the resulting *MappingSet* are composed of a polygon id, a polygon geometry, a rectangle geometry representing a group of consecutive input *raster* points and a boolean value which is `true` if the rectangle is within the polygon

Algorithm 5.4 *SpatialJoin_Contains_DatatypeQT* algorithm.

```
1: MSP ← MappingSet with input polygons
2: MSR ← MappingSet with input raster data
3: size ← spatial size of the input raster (from MSR)
4: resolution ← spatial resolution of the input raster (from MSR)
5: startXcoord ← integer x coordinate of the raster starting point (from MSR)
6: startYcoord ← integer y coordinate of the raster starting point (from MSR)
7: endXcoord ← integer x coordinate of the raster ending point (from MSR)
8: endYcoord ← integer y coordinate of the raster ending point (from MSR)
9: quadTree ← QuadTree(size, resolution)
10: quadTree.setSampling(startXcoord, startYcoord, endXcoord, endYcoord)
11: result ← array of resulting values initialized to null
12: for polygon in MSP do
13:     quadTreeCopy ← quadTree.copy()
14:     quadTreeCopy.insertGeometry(polygon)
15:     result.add(quadTreeCopy.processTree())
16: end for
17: return result
```

and `false` otherwise. The resulting *MappingSet* may have several values for each input polygon. Obviously, all the possible combinations of input polygons and input *raster* points have to be represented by the resulting combinations of polygons and rectangles.

A similar approach is taken by the operator *SpatialJoin_Contains_DimensionQT*. In this case, a resulting array containing the same elements as in the previous case is provided for each input polygon. An initial rectangle[7] is recursively divided in a Quad-Tree fashion, generating four new rectangles at each step. In this solution, the points of the input *raster* are ordered following the space filling curve depicted in Fig. 4.5. Similarly to the previous solution, a specific rectangle can not be divided anymore when 1) is completely inside the relevant polygon, 2) is completely outside the relevant polygon, or 3) its size reached the spatial resolution of the input *raster*. In the latter case, the rectangle is considered inside the polygon if its centroid is contained by the polygon, and considered outside otherwise.

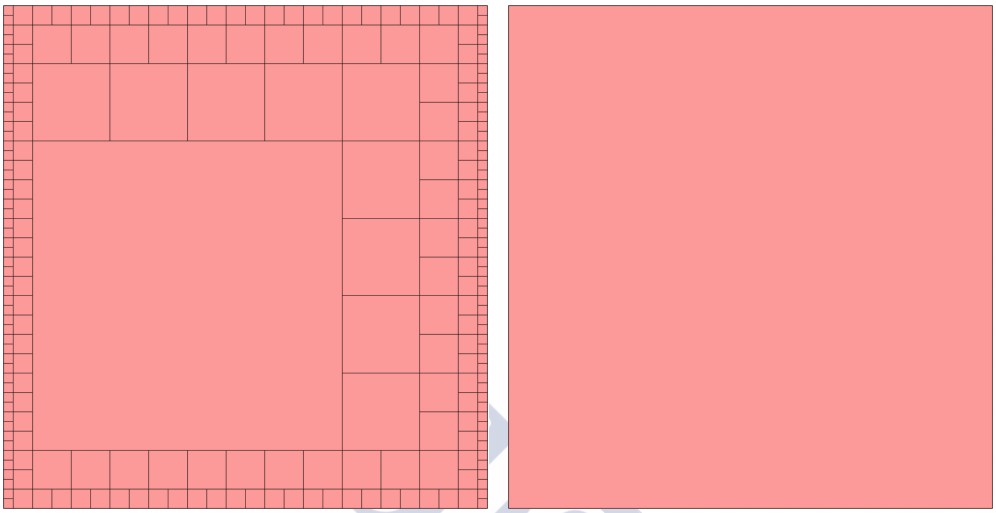
The main difference between these two approaches resides in the initial step. Whereas the operator *SpatialJoin_Contains_DimensionQT* starts to process polygons from a single rectangle covering the whole area of the input *raster* points, the operator *SpatialJoin_Contains_DatatypeQT* uses the whole area of the underlying data type (usually much bigger than the area

[7] The initial rectangle is generated to cover the whole area of the input *raster* data.

of the input *raster* data) to generate a Quad-Tree for the input *raster* data. Such an initial rectangle and initial Quad-Tree are respectively shown in Fig. 5.20(b) and Fig. 5.20(a) for an illustration example with a *raster* spatial resolution of 5000 meters. We can see that the border of the *raster* area is approximated by smaller rectangles in the *DatatypeQT* solution, providing more rectangles to be initially processed. Notice that all these rectangles will appear in the resulting array for each input polygon in addition to those rectangles required to process it. On the contrary, the resulting array of the *DimensionQT* solution for each input polygon only contains the rectangles required to process it. Fig. 5.21(a) shows the resulting rectangles returned by the *SpatialJoin_Contains_DatatypeQT* operator for a specific input polygon (depicted in blue). Rectangles in red are those returned as `false`, i.e., outside the input polygon, whereas rectangles inside the input polygon are colored in green. Fig. 5.21(b) depicts the resulting rectangles for the same input polygon returned by the *SpatialJoin_Contains_DimensionQT* operator.

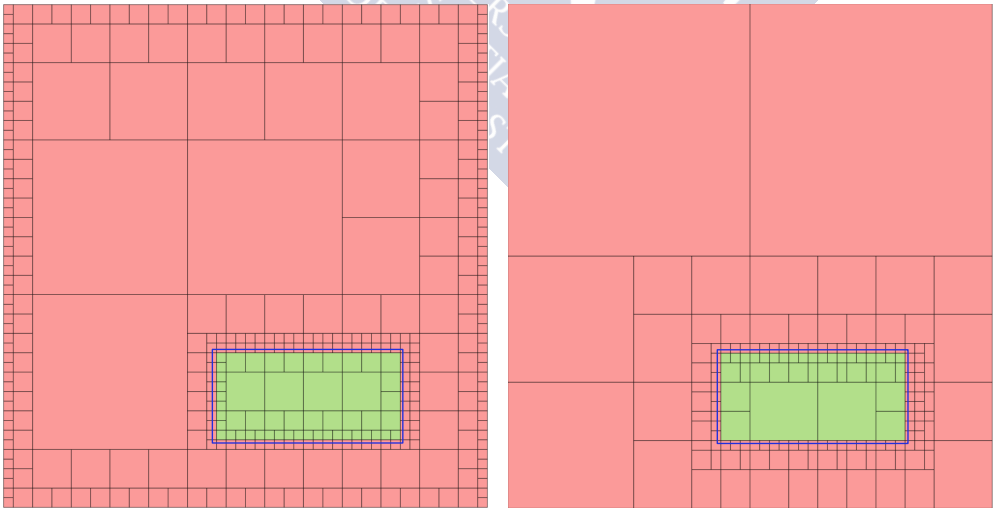
Execution times obtained by the operator *SpatialJoin_Contains_DatatypeQT* outperforms the best existing solutions. For small spatial resolutions (400-200 meters), Fig. 5.22(a) shows that the *DatatypeQT* solution obtains execution times similar to those obtained by LocationSpark, GeoSpark and Stark. The *DatatypeQT* solution enhances its performance as spatial resolution increases. For a spatial resolution of 12.5 meters, the *DatatypeQT* solution is more than one order of magnitude faster than the fastest remainder solution (LocationSpark). Furthermore, the *DatatypeQT* solution showed (Fig. 5.23(a)) a nice scalability performace, decreasing its execution time as the number of Spark executors increased.

The behavior of the *SpatialJoin_Contains_DimensionQT* showed a performance even better than the operator *SpatialJoin_Contains_DatatypeQT*. Execution times for different spatial resolutions showed an amazing outperformance, even improving the performance of the *DatatypeQT* solution. For small spatial resolutions, the *DimensionQT* solution is three times faster than the *DatatypeQT* solution (Fig. 5.22(a)). This difference increases as spatial resolution does. For the maximum resolution tested, *DimensionQT* is nearly seven times faster than *DatatypeQT*. For the tested spatial resolution, the *DimensionQT* solution showed an adequate scalability behavior (Fig. 5.23(a)), decreasing execution times as the number of Spark executors increases. Due to the great speed shown by the *DimensionQT* solution, a better scalability behaviour is expected as spatial resolution increases.



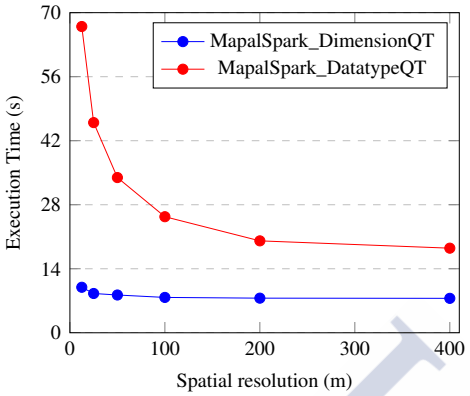
(a) *SpatialJoin_Contains_DatatypeQT* (b) *SpatialJoin_Contains_DimensionQT*

Figure 5.20: Initial rectangles for optimized *join* operators.



(a) *SpatialJoin_Contains_DatatypeQT* (b) *SpatialJoin_Contains_DimensionQT*

Figure 5.21: Resulting rectangles for optimized *join* operators.

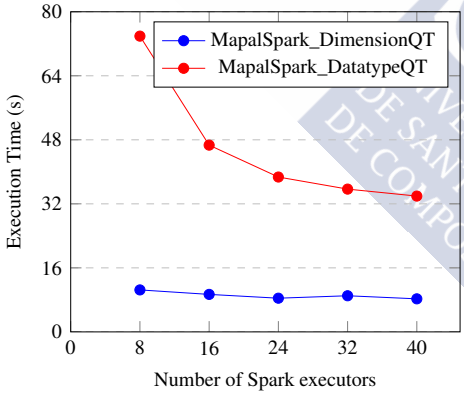


(a)

Spatial Resolution (m)	MapalSpark DimensionQT	MapalSpark DatatypeQT
400	7.53	18.50
200	7.57	20.10
100	7.73	25.38
50	8.26	33.94
25	8.59	45.96
12.5	9.95	66.97

(b)

Figure 5.22: Execution time for the optimized *spatial join* operators in a 40-executor cluster.



(a)

Number of Spark executors	MapalSpark DimensionQT	MapalSpark DatatypeQT
8	10.49	73.91
16	9.37	46.67
24	8.42	38.70
32	9.03	35.68
40	8.26	33.94

(b)

Figure 5.23: Execution time for the optimized *spatial join* operators with a spatial resolution of 50 meters.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

CHAPTER 6

CONCLUSIONS AND FUTURE RESEARCH

6.1 Conclusions

The design of the GeoDADIS framework is a generalization effort towards the development of data acquisition and dissemination servers. Heterogeneity in sensor data access and sensor data dissemination is a problem identified in data acquisition and monitoring research fields. GeoDADIS proposes a scalable and extensible architecture to solve it.

Flexibility features are achieved in GeoDADIS through the use of different software design patterns during the design of GeoDADIS components. The Adapter pattern eases the incorporation of new data services, remote control services and data acquisition channels with minimum changes in core components of the system. Incorporation of these elements only requires configuration data updates. Flexibility, scalability and extensibility features were validated during the development of a GeoDADIS based data acquisition and dissemination prototype that enables health monitoring in educational environments.

A framework for the analysis of spatio-temporal observation data, called SODA, was designed. Qualitative evaluation and comparison with related data management technologies and approaches were provided. First, based on a previously defined spatio-temporal data model, an observation data model was formalized. Then, a declarative spatio-temporal data analysis language was also described together with analytical processes and system operators. A prototype implementation is proposed and compared to state of the art solutions for spatial and spatio-temporal data analysis. Main advantages of the proposed framework may be summarized as follows.

- Both the spatial observation data model and the declarative definition of *Internal Processes* support and incorporate observation data semantics.
- Support for the integrated representation and analysis of both conventional E/R data and temporal, spatial and spatio-temporal sampled data is directly provided.
- The new defined temporal and spatial data types enable the representation and transformation between different data resolutions.
- The well known mathematical concept of *function* is used to represent both data (*Extensional MappingSets*) and behavior (*Intensional Mapppings*). Thus, this approach should be friendly to scientific users. Furthermore, a functional approach eases the definition and reuse of intermediate results.
- Incorporation of novel proposed languages, MAPAL and XODDL, in web services is simplified by their XML based nature.
- The efficient implementation of the framework leveraged the single non-nested data structure defined in the data model.

The benefits of defined data models and operators have been demonstrated by the performance results obtained by the implemented prototype. Execution times of the spatial join operation, implemented for comparison purposes, outperformed state of the art solutions for big spatial data analysis, e.g, GeoSpark, LocationSpark, Stark. Prototype execution times are orders of magnitude below the execution times obtained by competitors. Moreover, the prototype shows a scalability performance similar to the best performance solutions.

The main drawback of SODA is the adoption of a new functional data management paradigm by current DBMS users. However, the functional formalism has been combined with the well known logical formalism to define MAPAL. Thus, MAPAL constructors are very similar to those of current available languages like XQuery.

6.2 Future lines of research

The main future line of research on GeoDADIS is related to the support of remote sensors (e.g., lidar, radar, scatterometer, sounder) and complex measurements produced by them.

Regarding SODA, several future work issues may be identified.

- Incorporate query optimization techniques.
- Define new appropriate indexing structures.
- Design and implement new partitioning strategies for both *Dimensions* and *Extensional MappingSets*, and spatial data.
- Incorporation of approximate query processing techniques to stored *Extensional MappingSets*.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

APPENDIX A

PRIMITIVE MAPPINGS

Primitive mapping	Description
$and(b_1, b_2)$	Returns $b_1 \wedge b_2$.
$not(b)$	Returns $\bar{b}$.
$or(b_1, b_2)$	Returns $b_1 \vee b_2$.
$toCString(b)$	Returns a *CString* representation of b. If $b =$ `true`, "true" is returned. Otherwise, "false" is returned.

Table A.1: Description of primitive boolean mappings.

Primitive mapping	Description
$concat(str_1, str_2)$	Concatenates the *CString* str_2 to the end of the *CString* str_1.
$length(str)$	Returns the number of characters in str.
$lower(str)$	Converts all of the characters in str to lower case.
$upper(str)$	Converts all of the characters in str to upper case.
$toBoolean(str)$	Returns the *Boolean* value represented by str. If str equals to "true" (case insensitive), the boolean value `true` is returned. If str equals to "false" (case insensitive), the boolean value `false` is returned. Otherwise, an *IllegalFormatException* is thrown.

Primitive mapping	Description
$toDate(str)$	Returns the *Date* value represented by str. If str is not parsable as *Date*, an *IllegalFormatException* is thrown.
$toFixedPrecision(str)$	Returns the *FixedPrecision* value represented by str. *Precision* and *scale* parametric values are automatically extracted from str. If str does not contain a parsable *FixedPrecision* value, a *NumberFormatException* is thrown.
$toFixedPrecision(str, p, s)$	Returns the *FixedPrecision* value represented by str applying an *implicit* casting to *FixedPrecision(p,s)*. If str does not contain a parsable *FixedPrecision* value, a *NumberFormatException* is thrown.
$toInteger(str)$	Returns the *Integer* value represented by str. If str does not contain a parsable *Integer* value, a *NumberFormatException* is thrown.
$toReal(srt)$	Returns the *Real* value represented by str. If str does not contain a parsable *Real* value, a *NumberFormatException* is thrown.
$toTime(str)$	Returns the *Time* value represented by str. *Resolution* parametric value is automatically extracted from str. If str does not contain a parsable *Time* value, an *IllegalFormatException* is thrown.
$toTime(str, r)$	Returns the *Time* value represented by str applying an *implicit* casting to *Time(r)*. If str does not contain a parsable *Time* value, an *IllegalFormatException* is thrown.
$toTimeInstant(str)$	Returns the *TimeInstant* value represented by str. *Resolution* parametric value is automatically extracted from str. If str does not contain a parsable *TimeInstant* value, an *IllegalFormatException* is thrown.
$toTimeInstant(str, r)$	Returns the *TimeInstant* value represented by str applying an *implicit* casting to *TimeInstant(r)*. If str does not contain a parsable *TimeInstant* value, an *IllegalFormatException* is thrown.

Table A.2: Description of primitive string mappings.

Primitive mapping	**Description**
$abs(x)$	Returns the absolute value of x.
$acos(x)$	Returns the arc cosine of x. The returned angle is in the interval $[0, \pi]$.
$asin(x)$	Returns the arc sine of x. The returned angle is in the interval $[-\pi/2, \pi/2]$.
$atan(x)$	Returns the arc tangent of x. The returned angle is in the interval $[-\pi/2, \pi/2]$.
$atan2(x,y)$	Returns the θ component of the point (ρ, θ) in polar coordinates that corresponds to the point (x,y) in Cartesian coordinates.
$ceil(x)$	Returns the smallest (closest to negative infinity) *Integer* value that is greater than or equal to x.
$cos(x)$	Returns the trigonometric cosine of x interpreted as an angle in radians.
$divide(x,y)$	Returns the result of the division operation a/b.
$floor(x)$	Returns the largest (closest to positive infinity) *Integer* value that is less than or equal to x.
$ln(x)$	Returns the natural logarithm (base e) of x.
$log(x)$	Returns the base 10 logarithm of x.
$mod(x,y)$	Returns the remainder after the division of x by y.
$multiply(x,y)$	Returns the result of the multiplication operation $x \cdot y$.
$power(x,y)$	Returns x raised to the power of y.
$round(x)$	Returns x rounded to the nearest *Integer* value.
$round(x,y)$	Returns x rounded to the nearest *Real* value that have y decimal digits.
$sin(x)$	Returns the trigonometric sine of x interpreted as an angle in radians.
$sqrt(x)$	Returns the correctly rounded positive square root of x.
$subtract(x,y)$	Returns the result of the subtraction operation $x - y$.
$sum(x,y)$	Returns the result of the additive operation $x + y$.

Primitive mapping	Description
$tan(x)$	Returns the trigonometric tangent of x interpreted as an angle in radians.
$toBoolean(x)$	Returns the *Boolean* value `false` if $x = 0$ and returns `true` otherwise.
$toCString(x)$	Returns a *CString* representation of x.
$toFixedPrecision(x)$	Returns the *FixedPrecision* value that is equivalent to x. *Precision* and *scale* parametric values are automatically extracted from x.
$toFixedPrecision(x,p,s)$	Returns the *FixedPrecision* value that is equivalent x, applying an *implicit* casting to *FixedPrecision(p,s)*.
$toInteger(x)$	Equivalent to $floor(x)$.
$toPoint1D(x)$	Returns a *Point1D* value which spatial coordinate is equivalent to argument x. *Precision* and *resolution* parametric values are automatically extracted from a.
$toPoint1D(x,p,r)$	Returns a *Point1D* value which spatial coordinate is equivalent to argument x, applying an *implicit* casting to *Point1D(p,r)* data type.
$toReal(x)$	Returns the *Real* value that is equivalent to x.

Table A.3: Description of primitive numeric mappings.

Primitive mapping	Description
$subtract(t_1,t_2)$	If t_2 is a *Temporal* argument, returns an *Integer* value representing the number of *Temporal* values between t_2 and t_1. If t_2 is an *Integer* argument, returns the *Temporal* value resulting from subtracting t_2 *Temporal* values to t_1.
$subtractAsString(t_1,t_2)$	Returns $subtract(t_1,t_2)$ in a human readable format.
$sum(t_1,t_2)$	Returns the *Temporal* value resulting from adding t_2 *Temporal* values to t_1. Argument t_2 must be of type *Integer*.
$toCString(t_1)$	Returns a *CString* representation of t_1.
$toDate(t_1)$	Returns the *Date* value that is equivalent to t_1.

Primitive mapping	Description
$toTime(t_1)$	Returns the *Time* value that is equivalent to t_1. *Resolution* parametric value is automatically extracted from t_1.
$toTime(t_1, r)$	Returns the *Time* value that is equivalent to t_1, applying a casting to *Time(r)*.
$toTimeInstant(t_1)$	Returns the *TimeInstant* value that is equivalent to t_1. *Resolution* parametric value is automatically extracted from t_1.
$toTimeInstant(t_1, r)$	Returns the *TimeInstant* value that is equivalent to t_1, applying a casting to *TimeInstant(r)*.

Table A.4: Description of primitive temporal mappings.

Primitive mapping	Description
$subtract(p_1, p_2)$	If p_2 is a *Point1D* argument, returns an *Integer* value representing the distance (in number of *Point1D* values) between p_2 and p_1. If p_2 is an integer argument, returns the *Point1D* value resulting from subtracting p_2 *Point1D* values to p_1.
$sum(p_1, p_2)$	Returns the *Point1D* value resulting from adding p_2 *Point1D* values to p_1.
$toCString(p)$	Returns the *CString* representation of p.
$toFixedPrecision(p)$	Returns the *FixedPrecision* value that is equivalent to p. *Precision* and *scale* parametric values are automatically extracted from p.
$toFixedPrecision(p_1, p, s)$	Returns the *FixedPrecision* value that is equivalent to p_1, applying a casting to *FixedPrecision(p,s)*.
$toInteger(p)$	Returns the largest (closest to positive infinity) *Integer* value that is less than or equal to p.
$toPoint1D(p_1, p, r)$	Returns p_1 applying a casting to *Point1D(p,r)*.
$toReal(p)$	Returns the *Real* value that is equivalent to p.

Table A.5: Description of primitive Point1D mappings.

Primitive mapping	Description
$4neigh(p_1, p_2)$	Returns `true` if p_2 is within the 4-neighborhood of p_1, and returns `false` otherwise.
$8neigh(p_1, p_2)$	Returns `true` if p_2 is within the 8-neighborhood of p_1, and returns `false` otherwise.
$getPosition(p_1)$	Returns the position of p_1 in the underlying coordinate system according to the space filling curve shown in Fig. 4.5.
$getPrecision(p_1)$	Returns the precision of p_1.
$getResolution(p_1)$	Returns the resolution of p_1.
$getX(p_1)$	Returns the x coordinate of p_1: $n_x \cdot R$.
$getXint(p_1)$	Returns the integer x coordinate of p_1 in the underlying coordinate system: n_x.
$getY(p_1)$	Returns the y coordinate of p_1: $n_y \cdot R$.
$getYint(p_1)$	Returns the integer y coordinate of p_1 in the underlying coordinate system: n_y.
$shift(p_1, x, y)$	Returns the Point2D value $p_s = (n_x + x,\ n_y + y) \cdot R$, where $x, y \in \mathbb{Z}$ and R is the resolution of p_1.
$subtract(p_1, p_2)$	Returns an *Integer* value representing the distance (in number of *Point2D* values) between p_2 and p_1 according to the space filling curve shown in Fig. 4.5.
$toPoint2D(p_1, p, r)$	Returns p_1 applying a casting to *Point2D(p,r)*.

Table A.6: Description of primitive Point2D mappings.

Primitive mapping	Description
$buffer(g_1, d)$	Returns a *Geometry* value containing a buffer area around g_1 having the width d.
$contains(g_1, g_2)$	Returns `true` if g_1 contains g_2, and returns `false` otherwise.
$convexHull(g_1)$	Returns the smallest convex *Polygon* that contains all the points in g_1.

Primitive mapping	Description
$crosses(g_1, g_2)$	Returns `true` if g_1 crosses g_2, and returns `false` otherwise.
$difference(g_1, g_2)$	Returns a *Geometry* value representing the closure of the point-set composed of the points contained in g_1 that are not contained in g_2.
$disjoint(g_1, g_2)$	Returns `true` if g_1 is disjoint from g_2, and returns `false` otherwise.
$distance(g_1, g_2)$	Returns the minimum distance between g_1 and g_2.
$envelope(g_1)$	Returns a *Polygon* value representing the minimum bounding box of g_1.
$equals(g_1, g_2)$	Returns `true` if g_1 and g_2 are exactly equal, i.e., test whether the two geometries are *structurally equal*. Two geometries are exactly equal if and only if they have the same structure and they have the same values for their vertices (in exactly the same order).
$fromWkt(s)$	Reads a well-known text representation from the *CString* value s and returns the corresponding *Geometry* value.
$fromGml(s)$	Reads a GML2 representation from the *CString* value s and returns the corresponding *Geometry* value.
$getPrecision(g_1)$	Returns the precision of the underlying grid used by the point-set of g_1.
$getResolution(g_1)$	Returns the resolution of the underlying grid used by the point-set of g_1.
$gml(g_1)$	Returns g_1 as XML fragments in GML2 format.
$intersection(g_1, g_2)$	Returns a *Geometry* value representing the point-set common to both g_1 and g_2.
$intersects(g_1, g_2)$	Returns `true` if g_1 intersects g_2, and returns `false` otherwise.
$overlaps(g_1, g_2)$	Returns `true` if g_1 overlaps g_2, and returns `false` otherwise.

Primitive mapping	Description
$symDifference(g_1, g_2)$	Returns a *Geometry* value representing the closure of the point-set composed of the union of the points in g_1 which are not contained in g_2, with the points in g_2 not contained in g_1. It is actually a shortcut for $union(difference(g_1, g_2), difference(g_2, g_1))$.
$touches(g_1, g_2)$	`true` if g_1 touches g_2, and returns `false` otherwise.
$union(g_1, g_2)$	Returns a *Geometry* value representing the point-set of g_1 plus the point-set of g_2.
$within(g_1, g_2)$	Returns `true` if g_1 is within g_2, and returns `false` otherwise.
$wkt(g_1)$	Returns the well-known text representation of g_1.

Table A.7: Description of primitive mappings that are common to all geometry data types defined in SODA (including *Point2D(P,R)*).

Primitive mapping	Description
$endPoint(ls)$	Returns the last *Point2D* value within *ls*.
$isClosed(ls)$	Returns `true` if the first and last points of *ls* are exactly the same,and returns `false` otherwise.
$isRing(ls)$	Returns `true` if *ls* is both simple and closed, and returns `false` otherwise.
$isSimple(ls)$	Returns `true` if *ls* does not intersect itself, and returns `false` otherwise.
$length(ls)$	Returns the count of *Point2D* values within *ls*.
$startPoint(ls)$	Returns the first *Point2D* value within *ls*.

Table A.8: Description of primitive LineString mappings.

Primitive mapping	Description
$exterior(p)$	Returns a *LineString* value representing the exterior ring of p.
$holes(p)$	Returns the interior rings (aka “holes”) within p.

Primitive mapping	Description
$area(p)$	Returns the area of p.
$perimeter(p)$	Returns the perimeter of p.
$centroid(ls)$	Returns the centroid of p.

Table A.9: Description of primitive Polygon mappings.

Primitive mapping	Description
$voronoi(mp)$	Returns the Voronoi diagram of mp.

Table A.10: Description of primitive MultiPoint mappings.

Primitive mapping	Description
$isSimple(mls)$	Returns `true` if *mls* does not intersect itself, and returns `false` otherwise.

Table A.11: Description of primitive MultiLineString mappings.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

APPENDIX B

PUBLICATIONS

B.1 International Journals

[1] Sebastián Villarroya, José R.R. Viqueira, Manuel A. Regueiro, José A. Taboada, and José M. Cotos. SODA: A framework for spatial observation data analysis. *Distributed and Parallel Databases*, 34(1):65-99, Mar 2016.

[2] Sebastián Villarroya, José R.R. Viqueira, José M. Cotos, and Julián C. Flores. GeoDADIS: A framework for the development of geographic data acquisition and dissemination servers. *Computers & Geosciences*, 52:68-76, 2013.

B.2 International Conferences

[1] Diego Ferrón, Sebastián Villarroya, José R.R. Viqueira, and Tomás F. Pena. Towards Large Scale Environmental Data Processing with Apache Spark. In *Proceedings of the 20th Pacific Asia Conference on Information Systems* (PACIS), 2016.

[2] Sebastián Villarroya, David Martínez Casas, Moisés Vilar, José R.R. Viqueira, José A. Taboada, and José M. Cotos. Heterogeneous Sensor Data Integration for Crowdsensing Applications. In *Proceedings of the 18th International Database Engineering & Applications Symposium* (IDEAS), 2014.

[3] Sebastián Villarroya, José R.R. Viqueira, Manuel A. Regueiro, and José M. Cotos. Spatio-temporal Integrated Analysis with MAPAL. In *Proceedings of the 14th International Conference on Computational Science and Its Applications* (ICCSA), 2014.

[4] David Martínez Casas, Sebastián Villarroya, Moisés Vilar, José M. Cotos, José R.R. Viqueira, and José A. Taboada. Common Data Model in AmI Environments. In *Proceedings of the 8th International Conference on Ubiquitous Computing and Ambient Intelligence* (UCAmI), 2014.

[5] José R.R. Viqueira, David Martínez Casas, Sebastián Villarroya, and José A. Taboada. MappingSets for Spatial Observation Data Warehouses. In *Proceedings of the 3rd International Workshop on Information Management in Mobile Applications* (IMMoA), 2013.

[6] Sebastián Villarroya, Mª Jesús López-Otero, Luís Romero, José M. Cotos, and Víctor Pita. Modular and Scalable Multi-Interface Data Acquisition Architecture Design for Energy Monitoring in Fishing Vessels. In *Proceedings of the 10th International Work-Conference on Artificial Neural Networks* (IWANN), 2009.

B.3 National Conferences

[1] Diego Ferrón, Sebastián Villarroya, José R.R. Viqueira, and Tomás F. Pena. Procesamiento paralelo de datos medioambientales con Apache Spark. *V Congreso Español de Informática* (CEDI), 2016.

[2] Sebastián Villarroya, David Mera, Manuel A. Regueiro, and José M. Cotos. Diseño de Servidores de Adquisición y Publicación de Datos de Sensores. In *Proceedings de las XVII Jornadas de Ingeniería del Software y Bases de Datos* (JISBD), 2012.

[3] Sebastián Villarroya, Gabriel Álvarez, Roi Méndez, and José R.R. Viqueira. Análisis espacio-temporal en sistemas de bases de datos lógicos-funcionales. In *Proceedings de las XVII Jornadas de Ingeniería del Software y Bases de Datos* (JISBD), 2012.

B.4 Book Chapters

[1] Sebastián Villarroya, Mª Jesus López-Otero, José Varela Pet, and José M. Cotos. Chapter: Modular and Scalable Multi-Interface Data Acquisition Architecture Design for En-

ergy Monitoring in Fishing Vessels: an in-depth review. *AUTOMOBILES. Performance, Safety Assessment and Energy Consumption*, 2010.

B.5 Other Publications

[1] David Luaces, Sebastián Villarroya, Roi Méndez, José R.R. Viqueira, and José M. Cotos. Sistema para la extracción de entidades geográficas asociadas a telediarios: construcción de mapas para lengua castellana y gallega a través de HbbTV. In *Proceedings del VI Congreso de TV Digital Interactiva*, 2015.

[2] Roi Méndez, Julián Flores, Enrique Castelló-Mayo, Rubén Arenas, and Sebastián Villarroya. Sensorización Avanzada para Estudios Virtuales de Televisión. In *Proceedings del VI Congreso de TV Digital Interactiva*, 2015.

[3] Moisés Vilar, Sebastián Villarroya, José R.R. Viqueira, and José M. Cotos. GeoNews: Generación Automática de Contextos Geográficos para Programas de Noticias a través de HbbTV. In *Proceedings de las XX Jornadas de Ingeniería del Software y Bases de Datos* (JISBD), 2015.

[4] Manuel A. Regueiro, Sebastián Villarroya, Gabriel Sanmartín, and José R.R Viqueira. Integración de observaciones medioambientales: solución inicial y retos futuros. In *Proceedings de las XVII Jornadas de Ingeniería del Software y Bases de Datos* (JISBD), 2012.

[5] David Mera, José M. Cotos, Gabriel Álvarez, and Sebastián Villarroya. Detección de contaminantes en el medio marino a través ded análisis e interpretación de imágenes SAR utilizando técnicas de Inteligencia Artificial. In *Proceedings del XIV Congreso de la Asociación Española de Teledetección*, 2011.

[6] David Martínez Casas, José A. Taboada, Juan Enrique Arias Rodríguez, and Sebastián Villarroya. Anti-Icing Decision Support System based on a Multi-agent System and Data Mining. In *Proceedings of the International Symposium on Distributed Computing and Artificial Intelligence* (DCAI), 2011.

[7] Sebastián Villarroya, José M. Cotos, Guillermo Gómez, Borja Plaza, Manuel Fontán, Alexander Magdaleno, Xavier Vallvé, and Jaume Palou. Control, Monitoring and Data Acquisition Architecture Design for Clean Production of Hydrogen from Mini-Wind

Energy. In *Proceedings of the 5th European PV-Hybrid and Mini-Grid Conference*, 2010.

Bibliography

[1] Apache Parquet. `https://parquet.apache.org/`. Accessed: May 2018.

[2] D. Abadi, D.S. Myers, D.J. DeWitt, and S.R. Madden. Materialization Strategies in a Column-Oriented DBMS. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 466–475, April 2007.

[3] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 671–682, New York, NY, USA, 2006. ACM.

[4] OECD/IEA. International Energy Agency. *Capturing the Multiple Benefits of Energy Efficiency*. OECD Publishing, 2014.

[5] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. Hadoop GIS: A High Performance Spatial Data Warehousing System over Mapreduce. *Proc. VLDB Endow.*, 6(11):1009–1020, August 2013.

[6] Louai Alarabi and Mohamed F. Mokbel. A Demonstration of ST-hadoop: A MapReduce Framework for Big Spatio-temporal Data. *Proc. VLDB Endow.*, 10(12):1961–1964, August 2017.

[7] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, November 1983.

[8] Apache Cassandra. `http://cassandra.apache.org/`. Accessed: 2017-10-03.

[9] Apache Hadoop. http://hadoop.apache.org/. Accessed: 2017-11-15.

[10] Apache Spark. https://spark.apache.org/. Accessed: 2017-11-15.

[11] Peixe verde project. http://www.peixeverde.org/peixe_org_eng/index.htm. Accessed: 2017-11-15.

[12] Apache Hadoop YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html. Accessed: 2018-06-08.

[13] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[14] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[15] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. The Multidimensional Database System RasDaMan. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 575–577, New York, NY, USA, 1998. ACM.

[16] Peter Baumann and Sönke Holsten. A comparative analysis of array models for databases. In Tai-hoon Kim, Hojjat Adeli, Alfredo Cuzzocrea, Tughrul Arslan, Yanchun Zhang, Jianhua Ma, Kyo-il Chung, Siti Mariyam, and Xiaofeng Song, editors, *Database Theory and Application, Bio-Science and Bio-Technology: International Conferences, DTA and BSBT 2011, Held as Part of the Future Generation Information Technology Conference, FGIT 2001 in Conjunction with GDC 2011, Jeju Island, Korea, December 8-10, 2011. Proceedings*, pages 80–89. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[17] Fabio Bellifemine, Giancarlo Fortino, Roberta Giannantonio, Raffaele Gravina, Antonio Guerrieri, and Marco Sgroi. SPINE: a domain-specific framework for rapid

prototyping of WBSN applications. *Software: Practice and Experience*, 41(3):237–265, 2011.

[18] André B. Bondi. Characteristics of Scalability and Their Impact on Performance. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 195–203, New York, NY, USA, 2000. ACM.

[19] Mike Botts, George Percivall, Carl Reed, and John Davidson. OGC®Sensor Web Enablement: Overview and High Level Architecture. In Silvia Nittel, Alexandros Labrinidis, and Anthony Stefanidis, editors, *GeoSensor Networks*, pages 175–190. Springer-Verlag, Berlin, Heidelberg, 2008.

[20] Shawn Bowers, Joshua S. Madin, and Mark P. Schildhauer. A Conceptual Modeling Framework for Expressing Observational Data Semantics. In *Proceedings of the 27th International Conference on Conceptual Modeling*, ER '08, pages 41–54, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient Processing of Spatial Joins Using R-trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 237–246, New York, NY, USA, 1993. ACM.

[22] Paul G. Brown. Overview of sciDB: Large Scale Array Storage, Processing and Analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 963–968, New York, NY, USA, 2010. ACM.

[23] Arne Bröring, Christoph Stasch, and Johannes Echterhoff. Open Geospatial Consortium (OGC) Sensor Observation Service Interface Standard. `http://www.opengeospatial.org/standards/sos`, 2012. Accessed: 2017-10-04.

[24] João Pedro Cerveira Cordeiro, Gilberto Câmara, Ubirajara Moura De Freitas, and Felipe Almeida. Yet Another Map Algebra. *Geoinformatica*, 13(2):183–202, June 2009.

[25] Centro de Supercomputación de Galicia (CESGA) - Big Data Infrastructure. `https://www.cesga.es/en/infraestructuras/computacion/InfraBigDataEn`. Accessed: 2018-06-08.

[26] Avraam N. Chimaris and George A. Papadopoulos. Implementing a Generic Component-based Framework for Telecontrol Applications. *Softw. Pract. Exper.*, 37(10):1087–1132, August 2007.

[27] Eliseo Clementini and Paolino Di Felice. A model for representing topological relationships between complex geometric features in spatial databases. *Information Sciences*, 90(1):121 – 136, 1996.

[28] Matja Colnaric and Domen Verber. *Distributed Embedded Control Systems: Improving Dependability with Coherent Design*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[29] Michael Compton, Payam Barnaghi, Luis Bermudez, Raúl García-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, Vincent Huang, Krzysztof Janowicz, W. David Kelsey, Danh Le Phuoc, Laurent Lefort, Myriam Leggieri, Holger Neuhaus, Andriy Nikolov, Kevin Page, Alexandre Passant, Amit Sheth, and Kerry Taylor. The SSN Ontology of the W3C Semantic Sensor Network Incubator Group. *Web Semant.*, 17(C):25–32, December 2012.

[30] Simon Cox. Geographic Information - Observations and Measurements. Open Geospatial Consortium (OGC) Abstract Specification Topic 20. `http://www.opengeospatial.org/standards/om`, 2013. Accessed: 2017-10-03.

[31] Gianpaolo Cugola and Alessandro Margara. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[32] A. Daneels and W. Salter. What is SCADA? In *Accelerator and large experimental physics control systems. Proceedings, 7th International Conference, ICALEPCS'99, Trieste, Italy, October 4-8, 1999*, volume C991004, pages 339–343, 1999.

[33] Chris Date, Hugh Darwen, and Nikos A. Lorentzos. *Temporal Data and the Relational Model*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2002.

[34] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[35] A. Eldawy, M. F. Mokbel, S. Alharthi, A. Alzaidy, K. Tarek, and S. Ghani. SHAHED: A MapReduce-based system for querying and visualizing spatio-temporal satellite data. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1585–1596, April 2015.

[36] Ahmed. Eldawy and Mohamed F. Mokbel. Pigeon: A spatial MapReduce language. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1242–1245, March 2014.

[37] Ahmed. Eldawy and Mohamed F. Mokbel. SpatialHadoop: A MapReduce framework for spatial data. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1352–1363, April 2015.

[38] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. In *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, pages 407–416, June 2000.

[39] Raphael Ari Finkel and Jon Louis Bentley. Quad Trees a Data Structure for Retrieval on Composite Keys. *Acta Inf.*, 4(1):1–9, March 1974.

[40] Ixent Galpin, Christian Y. Brenninkmeijer, Alasdair J. Gray, Farhana Jabeen, Alvaro A. Fernandes, and Norman W. Paton. SNEE: A Query Processor for Wireless Sensor Networks. *Distrib. Parallel Databases*, 29(1-2):31–85, February 2011.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[42] GeoTIFF. `http://trac.osgeo.org/geotiff/`. Accessed: 2018-05-11.

[43] GeoTrellis: A Geographic Data Processing Engine for High Performance Applications. `https://geotrellis.io/`. Accessed: 2017-10-16.

[44] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association.

[45] Peter M. D. Gray. *The Functional Approach to Data Management: Modeling, Analyzing, and Integrating Heterogeneous Data*. SpringerVerlag, 2004.

[46] Ralf Hartmut Güting. Spatial Databases. In *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, Inc., 2001.

[47] Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, 25(1):1–42, 2000.

[48] Antonin Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[49] Morton Guy Macdonald. A computer oriented geodetic data base and a new technique in file sequencing. Technical Report Ottawa, Ontario, Canada, IBM Ltd., 1966.

[50] The Hadoop Distributed File System (HDFS). `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`. Accessed: 2018-02-14.

[51] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. MR-DBSCAN: A Scalable MapReduce-based DBSCAN Algorithm for Heavily Skewed Data. *Front. Comput. Sci.*, 8(1):83–99, February 2014.

[52] HiveQL Language Manual. `https://cwiki.apache.org/confluence/display/Hive/LanguageManual`. Accessed: 2017-10-19.

[53] Jeffery S. Horsburgh, David G. Tarboton, David R. Maidment, and Ilya Zaslavsky. Components of an environmental observatory information system. *Computers & Geosciences*, 37(2):207 – 218, 2011.

[54] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[55] IEC 61499-1:2012 Function blocks - Part 1: Architecture, 2012.

[56] IEC 61804-2:2006 Function blocks (FB) for process control - Part 2: Specification of FB concept, 2006.

[57] Instituto Geográfico Nacional - IGN. `http://www.ign.es/web/ign/portal`. Accessed: 2018-06-29.

[58] ISO/IEC 13249-3:2016. Information technology-Database languages-SQL multimedia and application packages-Part 3: Spatial., 2016. Accessed: 2017-10-04.

[59] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a Streaming SQL Standard. *Proc. VLDB Endow.*, 1(2):1379–1390, August 2008.

[60] JTS: Java Topology Suite. `https://github.com/locationtech/jts`. Accessed: 2017-11-15.

[61] Shannon P. Jackson Kevin A. Delin. Sensor web: a new instrument concept, 2001.

[62] H. R. Kolar, J. Cronin, P. Hartswick, A. C. Sanderson, J. S. Bonner, L. Hotaling, R. F. Ambrosio, Z. Liu, M. L. Passow, and M. L. Reath. Complex Real-time Environmental Monitoring of the Hudson River and Estuary System. *IBM J. Res. Dev.*, 53(3):378–387, May 2009.

[63] Eftichios Koutroulis and Kostas Kalaitzakis. Development of an integrated data-acquisition system for renewable energy sources systems monitoring. *Renewable Energy*, 28(1):139 – 152, 2003.

[64] Krishna Kulkarni and Jan-Eike Michels. Temporal Features in SQL:2011. *SIGMOD Rec.*, 41(3):34–43, October 2012.

[65] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. YSmart: Yet Another SQL-to-MapReduce Translator. In *2011 31st International Conference on Distributed Computing Systems*, pages 25–36, June 2011.

[66] Zhenlong Li, Fei Hu, John L. Schnase, Daniel Q. Duffy, Tsengdar Lee, Michael K. Bowen, and Chaowei Yang. A spatiotemporal indexing approach for efficient processing of big array-based climate data with MapReduce. *International Journal of Geographical Information Science*, 31(1):17–35, 2017.

[67] Steve H.L. Liang, Arie Croitoru, and C. Vincent Tao. A distributed geospatial infrastructure for sensor web. *Computers & Geosciences*, 31(2):221 – 231, 2005. Geospatial Research in Europe: AGILE 2003.

[68] Nikos A. Lorentzos and Jose Ramon Rios Viqueira. Relational Formalism for the Management of Spatial Data. *Comput. J.*, 49(1):62–81, 2006.

[69] J. Lu and R. H. Güting. Parallel Secondo: Boosting Database Engines with Hadoop. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 738–743, Dec 2012.

[70] Qiang Ma, Bin Yang, Weining Qian, and Aoying Zhou. Query Processing of Massive Trajectory Data Based on Mapreduce. In *Proceedings of the First International Workshop on Cloud Data Management*, CloudDB '09, pages 9–16, New York, NY, USA, 2009. ACM.

[71] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.*, 30(1):122–173, March 2005.

[72] Joshua Madin, Shawn Bowers, Mark Schildhauer, Serguei Krivov, Deana Pennington, and Ferdinando Villa. An ontology for describing and synthesizing ecological observation data. *Ecological Informatics*, 2(3):279 – 296, 2007. Meta-information systems and ontologies. A Special Feature from the 5th International Conference on Ecological Informatics ISEI5, Santa Barbara, CA, Dec. 4-7, 2006.

[73] Magellan: Geospatial Analytics Using Spark. `https://github.com/harsha2010/magellan`. Accessed: 2017-10-18.

[74] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, June 2011.

[75] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, January 2016.

[76] R. L. Miller, K. Davis, O. von Zweck, V. G. Sprague, and J. Sommers. Shipboard data acquisition, processing and analysis: an integrated software approach. In *OCEANS '95. MTS/IEEE. Challenges of Our Changing Global Environment. Conference Proceedings.*, volume 3, pages 1794–1799 vol.3, Oct 1995.

[77] Mongo DB. `https://www.mongodb.com/`. Accessed: 2017-10-03.

[78] J. M. Moore, J. S. Charters, and C. de Moustier. Multi-sensor real-time data acquisition and preprocessing at sea. In *OCEANS '88. A Partnership of Marine Interests. Proceedings*, pages 509–517 vol.2, Oct 1988.

[79] Razvan Musaloiu-Elefteri, Andreas Terzis, Katalin Szlavecz, Alexander S. Szalay, Joshua Cogan, and Jim Gray. Life under your feet: A wireless soil ecology sensor network. In *Proceedings of 3rd IEEE Workshop on Embedded Networked Sensors, EmNets 2006 2005*, 2006.

[80] Philippe Naveau and Denis Allard. Modeling skewness in spatial data analyis without data transformation. In Oy Leuangthong and Clayton V. Deutsch, editors, *Geostatistics Banff 2004*, pages 929–937. Springer Netherlands, Dordrecht, 2005.

[81] Markus Neteler and Helena Mitasova. *Open Source GIS: A GRASS GIS Approach*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[82] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *2011 IEEE*

12th International Conference on Mobile Data Management, volume 1, pages 7–16, June 2011.

[83] Regina O. Obe and Leo S. Hsu. *PostGIS in Action*. Manning Publications Co., Greenwich, CT, USA, 2nd edition, 2015.

[84] Open Geospatial Consortium (OGC): OpenGIS Sensor Model Language (SensorML) Implementation Specification. `http://www.opengeospatial.org/standards/sensorml`, 2007. Accessed: 2017-10-04.

[85] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[86] Jack A. Orenstein. Spatial query processing in an object-oriented database system. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 326–336, New York, NY, USA, 1986. ACM.

[87] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest Neighbor Queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 71–79, New York, NY, USA, 1995. ACM.

[88] Peter Schut. Open Geospatial Consortium (OGC): OpenGIS Web Processing Service (WPS). `http://www.opengeospatial.org/standards/wps`, 2007. Accessed: 2017-10-04.

[89] Christian Schwab, Marcus Tangermann, and Luca Ferrarini. Web based Methodology for Engineering and Maintenance of Distributed Control Systems: The TORERO Approach. In *Proceedings of 3rd IEEE International Conference on Industrial Informatics, INDIN 2005*, pages 32–37, 2005.

[90] Shashi Shekhar, Steven K. Feiner, and Walid G. Aref. Spatial Computing. *Commun. ACM*, 59(1):72–81, December 2015.

[91] Richard Thomas Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Norwell, MA, USA, 1995.

[92] Stefan Hagedorn and Timo Räth. Efficient spatio-temporal event processing with STARK. In *20th International Conference on Extending Database Technology*, 2017.

[93] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[94] Haoyu Tan, Wuman Luo, and Lionel M. Ni. CloST: A Hadoop-based Storage System for Big Spatio-temporal Data Analytics. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 2139–2143, New York, NY, USA, 2012. ACM.

[95] Mingjie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. LocationSpark: A Distributed In-memory Data Management System for Big Spatial Data. *Proc. VLDB Endow.*, 9(13):1565–1568, September 2016.

[96] Kleanthis Thramboulidis. Development of Distributed Industrial Control Applications: The CORFU Framework. In *Proceedings of 4th IEEE International Workshop on Factory Communications Systems*, pages 39–46, 2002.

[97] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution over a Map-reduce Framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[98] Alejandro A. Vaisman and Esteban Zimányi. A multidimensional model representing continuous fields in spatial data warehouses. In Divyakant Agrawal, Walid G. Aref, Chang-Tien Lu, Mohamed F. Mokbel, Peter Scheuermann, Cyrus Shahabi, and Ouri Wolfson, editors, *17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2009, November 4-6, 2009, Seattle, Washington, USA, Proceedings*, pages 168–177. ACM, 2009.

[99] Vertica. `https://www.vertica.com/`. Accessed: 2017-10-03.

[100] S. Villarroya, J.R.R. Viqueira, J.M. Cotos, and J.C. Flores. GeoDADIS: A framework for the development of geographic data acquisition and dissemination servers. *Computers & Geosciences*, 52:68 – 76, 2013.

[101] Sebastian Villarroya, David Martínez Casas, Moisés Vilar, José R. Ríos Viqueira, José A. Taboada, and José M. Cotos. Heterogeneous Sensor Data Integration for Crowdsensing Applications. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, IDEAS '14, pages 270–273, New York, NY, USA, 2014. ACM.

[102] Sebastián Villarroya, Ma. Jesús Otero, Luís Romero, José M. Cotos, and Víctor Pita. Modular and Scalable Multi-interface Data Acquisition Architecture Design for Energy Monitoring in Fishing Vessels. In *Proceedings of the 10th International Work-Conference on Artificial Neural Networks: Part II: Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, IWANN '09, pages 531–538, Berlin, Heidelberg, 2009. Springer-Verlag.

[103] Sebastián Villarroya, José R. R. Viqueira, Manuel A. Regueiro, José A. Taboada, and José M. Cotos. SODA: A framework for spatial observation data analysis. *Distributed and Parallel Databases*, 34(1):65–99, Mar 2016.

[104] Jose R. Rios Viqueira and Nikos A. Lorentzos. SQL Extension for Spatio-temporal Data. *The VLDB Journal*, 16(2):179–200, April 2007.

[105] VoltDB. `https://www.voltdb.com/`. Accessed: 2017-10-03.

[106] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. Simba: Efficient In-Memory Spatial Analytics. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1071–1085, New York, NY, USA, 2016. ACM.

[107] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in Cloud. In *2015 31st IEEE International Conference on Data Engineering Workshops*, pages 34–41, April 2015.

[108] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. GeoSpark: A Cluster Computing Framework for Processing Large-scale Spatial Data. In *Proceedings of the 23rd*

SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL '15, pages 70:1–70:4, New York, NY, USA, 2015. ACM.

[109] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[110] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[111] Ying Zhang, Martin Kersten, and Stefan Manegold. SciQL: Array Data Processing Inside an RDBMS. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1049–1052, New York, NY, USA, 2013. ACM.

[112] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. SCOPE: Parallel Databases Meet MapReduce. *The VLDB Journal*, 21(5):611–636, October 2012.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

List of Figures

List of Tables

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA